Controle de Versão com Subversion

Para Subversion 1.4

(Compilado da revisão 311)

Ben Collins-Sussman
Brian W. Fitzpatrick
C. Michael Pilato

Controle de Versão com Subversion: Para Subversion 1.4: (Compilado da revisão 311)

por Ben Collins-Sussman, Brian W. Fitzpatrick, e C. Michael Pilato

Publicado (TBA)

Índice

Lista de Figuras

Lista de Tabelas

Lista de Exemplos

Apresentação

Karl Fogel
Chicago, 14 de Março de 2004

Uma base ruim de Perguntas Freqüentes (FAQ), é aquela que é composta não de perguntas que as pessoas realmente fizeram, mas de perguntas que o autor da FAQ *desejou* que as pessoas tivessem feito. Talvez você já tenha visto isto antes:

> P: De que forma posso utilizar o Glorbosoft XYZ para maximizar a produtividade da equipe?
>
> R: Muitos dos nossos clientes desejam saber como podem maximizar a produtividade através de nossas inovações patenteadas de groupware para escritórios. A resposta é simples: primeiro, clique no menu “`Arquivo`”. Desça até a opção “`Aumentar Produtividade`”, então…

O problema com estas bases de FAQ é que eles não são, propriamente ditas, FAQ. Ninguém nunca liga para o suporte técnico e pergunta “Como nós podemos maximizar a produtividade?”. Em vez disso, as pessoas fazem perguntas muito mais específicas, como: “Como podemos alterar o sistema de calendário para enviar lembretes dois dias antes ao invés de um?”, etc. Mas é muito mais fácil forjar Perguntas Freqüentes imaginárias do que descobrir as verdadeiras. Compilar uma verdadeira base de FAQ exige um esforço contínuo e organizado: através do ciclo de vida do software, as questões que chegam devem ser rastreadas, respostas monitoradas, e tudo deve ser reunido em um todo coerente, pesquisável que reflete a experiência coletiva dos usuários em seu mundo. Isto requer a paciência e a atitude observadora de um cientista. Nenhuma grande teoria ou pronunciamentos visionários aqui—olhos abertos e anotações precisas são o principal.

O que eu amo neste livro é que ele surgiu deste processo, e isto é evidenciado em cada página. Ele é o resultado direto dos encontros entre os autores e usuários. Ele começou quando Ben Collins-Sussman observou que as pessoas estavam fazendo as mesmas perguntas básicas diversas vezes nas listas de discussão do Subversion, como por exemplo: Quais são os procedimentos-padrão para se utilizar o Subversion? Branches e tags funcionam do mesmo modo como em outros sistemas de controle de versão? Como eu posso saber quem realizou uma alteração em particular?

Frustrado em ver as mesmas questões dia após dia, Ben trabalhou intensamente durante um mês no verão de 2002 para escrever *The Subversion Handbook*, um manual de sessenta páginas cobrindo todas as funcionalidades básicas do Subversion. O manual não tinha a pretensão de ser completo, mas ele foi distribuído com o Subversion e auxiliou os usuários a ultrapassarem as dificuldades iniciais na curva de aprendizado. Quando a O'Reilly and Associates decidiu publicar um livro completo sobre o Subversion, o caminho menos crítico estava óbvio: apenas estender o manual.

Para os três co-autores do novo livro então lhes foi dada uma oportunidade ímpar. Oficialmente, sua tarefa era escrever um livro numa abordagem top-down, começando pelo sumário e seu esboço inicial. Mas eles também tinham acesso a um fluxo constante—de fato, um gêiser incontrolável — de conteúdo de origem bottom-up. O Subversion já esteve nas mãos de centenas de usuários anteriores, que geraram toneladas de feedback, não apenas sobre o Subversion, mas também sobre sua documentação existente.

Durante todo o tempo em que eles escreveram o livro, Ben, Mike, e Brian habitaram as listas de discussão e salas de bate-papo do Subversion, registrando cuidadosamente os problemas que os usuário estavam tendo em situações na vida-real. Monitorar este feedback, fazia parte da descrição de sua função na CollabNet, e isto lhes deu uma enorme vantagem quando começaram a documentar o Subversion. O livro que eles produziram é solidamente fundamentado na rocha da experiência, não nas areias mutáveis da ilusão; ele combina os melhores aspectos de um manual de usuário e uma base de FAQ. Esta dualidade talvez não seja perceptível numa primeira leitura. Lido na ordem, de frente para trás, o livro é uma descrição bem direta de uma peça de software. Existe a visão geral, o obrigatório tour, o capítulo sobre configuração administrativa, alguns tópicos avançados, e é claro uma referência de comandos e um guia de resolução de problemas. Somente quando você o ler novamente

mais tarde, procurando a solução para um problema específico, é que sua autenticidade reluzirá: os detalhes telling que só podem advir de um encontro com o inusitado, os exemplos surgidos de casos de utilização reais, e muito de toda a sensibilidade das necessidades e dos pontos de vista do usuário.

É claro, que ninguém pode prometer que este livro responderá todas as dúvidas que você tem sobre o Subversion. Certas vezes, a precisão com que ele antecipa suas perguntas parecerá assustadoramente telepática; ainda sim, ocasionalmente, você vai tropeçar em alguma falha no conhecimento da comunidade, e sairá de mão vazias. Quando isto acontecer, a melhor coisa que você pode fazer é enviar um email para `<users@subversion.tigris.org>` e apresentar seu problema. Os autores ainda estão lá, continuam observando, e não somente os três listados na capa, mas muitos outros que contribuíram com correções e materiais originais. Do ponto de vista da comunidade, resolver o seu problema é meramente um agradável efeito de um projeto muito maior—realmente, o ajuste paulatino deste livro, e em último caso, do próprio Subversion, para ver mais de perto como as pessoas o utilizam. Eles estão ansiosos para ouvir você não apenas porque eles querem ajudá-lo, mas porque você também os ajuda. Com o Subversion, assim como em todo projeto ativo de software livre, *você não está sozinho*.

Que este livro seja seu primeiro companheiro.

Prefácio

> “É importante não deixar que o perfeito se torne inimigo do bom, mesmo quando você puder estar certo sobre o que o perfeito é. Duvide quando você não puder. Como é desagradável ser aprisionado pelos erros passados, você não pode fazer qualquer progresso tendo medo de sua própria sombra durante a ação.”
>
> —Greg Hudson

No mundo dos softwares open-source, o Concurrent Versions System (CVS) foi a ferramenta escolhida para controle de versão por muitos anos. E com razão. O próprio CVS é um software open-source também, e seu modus operandi não-restritivo e o suporte a operações de rede permitiram que diversos programadores distribuídos geograficamente compartilhassem seus trabalhos. Ele atende à natureza colaborativa do mundo open-source como um todo. O CVS e seu modelo de desenvolvimento semi-caótico se tornou um marco da cultura open-source.

Mas o CVS também tinha seus defeitos, e simplesmente corrigir estes defeitos prometia ser um enorme esforço. Chega o Subversion. Desenvolvido para ser um sucessor do CVS, os criadores do Subversion pretendiam ganhar a simpatia dos usuários CVS de duas maneiras—criando um sistema open-source com o projeto (e a “aparência”) semelhante ao do CVS, e tentando evitar muitos de seus conhecidos defeitos. Por mais que o resultado não seja necessariamente a próxima grande evolução no projeto de controle de versão, o Subversion *é* muito poderoso, muito usável, e muito flexível. E agora muitos projetos open-source, quase todos recém-iniciados, preferem agora o Subversion ao CVS.

Este livro foi escrito para documentar a série 1.4 do sistema de controle de versão Subversion. Nós tentamos ser bem profundos em nossa abordagem. Entretanto, o Subversion possui uma comunidade de desenvolvedores próspera e cheia de energia, então eles já têm um conjunto de recursos e melhorias planejadas para futuras versões do Subversion que podem mudar alguns dos comandos e notas específicas deste livro.

Público-Alvo

Este livro foi escrito para pessoas habituadas com computadores que desejam usar o Subversion para gerenciar seus dados. Ainda que o Subversion rode em vários sistemas operacionais diferentes, a sua interface de usuário primária é baseada em linha de comando. Essa ferramenta de linha de comando (**svn**), e alguns programas auxiliares, são o foco deste livro.

Por motivo de consistência, os exemplos neste livro presumem que o leitor esteja usando um sistema operacional baseado em Unix e se sinta relativamente confortável com Unix e com interfaces de linha de comando. Dito isto, o programa **svn** também roda em plataformas não-Unix, como o Microsoft Windows. Com poucas exceções, como o uso de barras invertidas (`\`) em lugar de barras regulares (`/`) para separadores de caminho, a entrada e a saída desta ferramenta, quando executada no Windows, são idênticas às do seu companheiro Unix.

Muitos leitores são provavelmente programadores ou administradores de sistemas que necessitam rastrear alterações em código-fonte. Essa é a finalidade mais comum para o Subversion, e por isso é o cenário por trás de todos os exemplos deste livro. Entretanto, o Subversion pode ser usado para gerenciar qualquer tipo de informação—imagens, músicas, bancos de dados, documentação, etc. Para o Subversion, dados são apenas dados.

Enquanto que este livro presume que o leitor nunca usou um sistema de controle de versão, nós também tentamos facilitar para os usuários do CVS (e outros sistemas) a migração para o Subversion. Ocasionalmente, notas especiais poderão mencionar outros controles de versão. Há também um apêndice que resume muitas das diferenças entre CVS e Subversion.

Note também que os exemplos de código-fonte usados ao longo do livro são apenas exemplos. Ainda que eles compilem com os truques apropriados do compilador, seu propósito é ilustrar um cenário em particular, não necessariamente servir como exemplos de boas práticas de programação.

Como Ler Este Livro

Livros técnicos sempre enfrentam um certo dilema: se os leitores devem fazer uma leitura *top-down* (do início ao fim) ou *bottom-up* (do fim ao começo). Um leitor top-down prefere ler ou folhear toda a documentação antes, ter uma visão geral de como o sistema funciona e apenas, então, é que que ele inicia o uso do software. Já um leitor bottom-up "aprende fazendo", ele é alguém que mergulha no software e o esmiuça, voltando às seções do livro quando necessário. Muitos livros são escritos para um ou outro tipo de pessoa, e esse livro é, sem dúvida, indicado para o leitor top-down. (Se você está lendo esta seção, provavelmente você já é um leitor top-down nato). No entanto, se você é um leitor bottom-up, não se desespere. Mesmo que este livro pode ser definido como um apanhado geral sobre o Subversion, o conteúdo de cada seção tende a aprofundar com exemplos específicos nos quais você pode aprender fazendo. As pessoas impacientes que já querem ir fazendo, podem pular direto para o Apêndice A, *Guia Rápido de Introdução ao Subversion*.

Independente do seu estilo de aprendizado, este livro pretende ser útil para os mais diversos tipos de pessoas — os que não possuem nenhuma experiência com controle de versão até os administradores de sistema mais experientes. Dependendo do seu conhecimento, certos capítulos podem ser mais ou menos importantes para você. A lista abaixo é uma "recomendação de leitura" para os diversos tipos de leitores:

Administradores de Sistemas Experientes
: Supõe-se aqui que você, provavelmente, já tenha usado controle de versão anteriormente, e está morrendo de vontade de usar um servidor Subversion o quanto antes. O Capítulo 5, *Administração do Repositório* e o Capítulo 6, *Configuração do Servidor* irão mostrar como criar seu primeiro repositório e torná-lo disponível na rede. Depois disso, o Capítulo 2, *Uso Básico* e o Apêndice B, *Subversion para Usuários de CVS* vão mostrar o caminho mais rápido para se aprender a usar o cliente Subversion.

Novos usuário
: Seu administrador provavelmente já disponibilizou um servidor Subversion, e você precisa aprender a usar o cliente. Se você nunca utilizou um sistema de controle de versão, então o Capítulo 1, *Conceitos Fundamentais* será vital para introduzir e mostrar as idéias por trás do controle de versão. O Capítulo 2, *Uso Básico* é um guia do cliente do Subversion.

Usuários avançados
: Seja você um usuário ou um administrador, eventualmente seu projeto irá crescer muito. Você irá querer aprender a fazer coisas avançadas com o Subversion, como usar ramos e fazer fusões (Capítulo 4, *Fundir e Ramificar*), como usar as propriedades de suporte do Subversion (Capítulo 3, *Tópicos Avançados*), como configurar as opções de tempo de execu o (Capítulo 7, *Customizando sua Experiência com Subversion*), entre outras coisas. Estes capítulos não são críticos no início, porém não deixe de lê-los quando estiver familiarizado com o básico.

Desenvolvedores
: Presumidamente, você já está familiarizado com o Subversion, e quer ou extendê-lo ou construir um novo software baseado nas suas diversas APIs. O Capítulo 8, *Incorporando o Subversion* foi feito justamente pra pra você.

O livro termina com um material de referência — o Capítulo 9, *Referência Completa do Subversion* é um guia de referência para todos os comandos do Subversion, e os apêndices cobrem um número considerável de tópicos úteis. Estes capítulos serão os que você irá voltar com mais freqüência depois que terminar de ler o livro.

Convenções Usadas Neste Livro

Esta seção cobre as várias convenções usadas neste livro.

Convenções tipográficas

Largura constante
: Comandos usados, comando de saída, e opções

`Largura constante em itálico`
: Usado para substituir itens no código e texto

`Itálico`
: Usado para nomes de arquivo e diretório

Ícones

Este ícone representa uma nota relacionada ao texto citado.

Este ícone representa uma dica útil relacionada ao texto citado.

Este ícone representa um aviso relacionado ao texto citado.

Organização Deste Livro

Abaixo estão listados os capítulos e seus conteúdos estão listados:

Prefácio
: Cobre a história do Subversion bem como suas características, arquitetura e compomentes.

Capítulo 1, *Conceitos Fundamentais*
: Explica o básico sobre controle de versão e os diferentes modelos de versionamento, o repositório do Subversion, cópias de trabalho e revisões.

Capítulo 2, *Uso Básico*
: Faz um tour por um dia na vida de um usuário do Subversion. Demonstra como usar o cliente do Subversion para obter, modificar e submeter dados.

Capítulo 3, *Tópicos Avançados*
: Cobre as características mais complexas que os usuários regulares irão encontrar eventualmente, como metadados, travamento de arquivo(*locking*) e rotulagem de revisões.

Capítulo 4, *Fundir e Ramificar*
: Discute sobre ramos, fusões, e rotulagem, incluindo as melhores práticas para a criação de ramos e fusões, casos de uso comuns, como desfazer alterações, e como facilitar a troca de um ramo para o outro.

Capítulo 5, *Administração do Repositório*
: Descreve o básico de um repositório Subversion, como criar, configurar, e manter um repositório, e as ferramentas que podem ser usadas para isso.

Capítulo 6, *Configuração do Servidor*
: Explica como configurar um servidor Subversion e os diferentes caminhos para acessar seu repositório: `HTTP`, o protocolo`svn` e o acesso local pelo disco. Também cobre os detalhes de autenticação, autorização e acesso anônimo.

Capítulo 7, *Customizando sua Experiência com Subversion*
: Explora os arquivos de configuração do cliente Subversion, a internacionalização de texto, e como fazer com que ferramentas externas trabalhem com o subversion.

Capítulo 8, *Incorporando o Subversion*
: Descreve as partes internas do Subversion, o sistema de arquivos do Subversion, e a cópia de trabalho da área administrativa do ponto de vista de um programador. Demonstra como usar as APIs públicas para escrever um programa que usa o Subversion, e o mais importante, como contribuir para o desenvolvimento do Subversion.

Capítulo 9, *Referência Completa do Subversion*
: Explica em detalhes todos os subcomandos do **svn**, **svnadmin**, e **svnlook** com abundância de exemplos para toda a família!

Apêndice A, *Guia Rápido de Introdução ao Subversion*
: Para os impacientes, uma rápida explicação de como instalar o Subversion e iniciar seu uso imediatamente. Você foi avisado.

Apêndice B, *Subversion para Usuários de CVS*
: Cobre as similaridades e diferenças entre o Subversion e o CVS, com numerosas sugestões de como fazer para quebrar todos os maus hábitos que você adquiriu ao longo dos anos com o uso do CVS. Inclui a descrição de número de revisão do Subversion, versionamento de diretórios, operações offline, **update** vs. **status**, ramos, rótulos, resolução de conflitos e autenticação.

Apêndice C, *WebDAV e Autoversionamento*
: Descreve os detalhes do WebDAV e DeltaV, e como você pode configurar e montar seu repositório Subversion em modo de leitura/escrita em um compartilhamento DAV.

Apêndice D, *Ferramentas de Terceiros*
: Discute as ferramentas que suportam ou usam o Subversion, incluindo programas clientes alternativos, ferramentas para navegação no repositório, e muito mais.

Este Livro é Livre

Este livro teve início com a documentação escrita pelos desenvolvedores do projeto Subversion, a qual foi reunida em um único trabalho e reescrito. Assim, ele sempre esteve sob uma licença livre. (Veja Apêndice E, *Copyright*.) De fato, o livro foi escrito sob uma visão pública, originalmente como parte do próprio projeto Subversion. Isto significa duas coisas:

- Você sempre irá encontrar a última versão deste livro no próprio repositório Subversion do livro.
- Você pode fazer alterações neste livro e redistribuí-lo, entretanto você deve fazê-lo—sob uma licença livre. Sua única obrigação é manter o créditos originais dos autores. É claro que, ao invés de distribuir sua própria versão deste livro, gostaríamos muito mais que você enviasse seu feedback e correções para a comunidade de desenvolvimento do Subversion.

O site deste livro está em desenvolvimento, e muitos dos tradutores voluntários estão se reunindo no site http://svnbook.red-bean.com. Lá você pode encontrar links para as últimas versões lançadas e versões compiladas deste livro em diversos formatos, bem como as instruções de acesso ao repositório Subversion do livro(onde está o código-fonte em formato DocBook XML) Um feedback é bem vindo —e encorajado também. Por favor, envie todos os seus comentários, reclamações, e retificações dos fontes do livro para o e-mail `<svnbook-dev@red-bean.com>`.

Agradecimentos

Este livro não existiria (nem seria útil) se o Subversion não existisse. Assim, os autores gostariam de agradecer ao Brian Behlendorf e à CollabNet, pela visão em acreditar em um arriscado e ambicioso

projeto de Código Aberto; Jim Blandy pelo nome e projeto original do Subversion—nós amamos você, Jim; Karl Fogel, por ser um excelente amigo e grande líder na comunidade, nesta ordem.[1]

Agradecimentos a O'Reilly e nossos editores, Linda Mui e Tatiana Diaz por sua paciente e apoio.

Finalmente, agrademos às inúmeras pessoas que contribuíram para este livro com suas revisões informais, sugestões e retificações. Certamente, esta não é uma lista completa, mas este livro estaria incompleto e incorreto sem a ajuda de: David Anderson, Jani Averbach, Ryan Barrett, Francois Beausoleil, Jennifer Bevan, Matt Blais, Zack Brown, Martin Buchholz, Brane Cibej, John R. Daily, Peter Davis, Olivier Davy, Robert P. J. Day, Mo DeJong, Brian Denny, Joe Drew, Nick Duffek, Ben Elliston, Justin Erenkrantz, Shlomi Fish, Julian Foad, Chris Foote, Martin Furter, Dave Gilbert, Eric Gillespie, David Glasser, Matthew Gregan, Art Haas, Eric Hanchrow, Greg Hudson, Alexis Huxley, Jens B. Jorgensen, Tez Kamihira, David Kimdon, Mark Benedetto King, Andreas J. Koenig, Nuutti Kotivuori, Matt Kraai, Scott Lamb, Vincent Lefevre, Morten Ludvigsen, Paul Lussier, Bruce A. Mah, Philip Martin, Feliciano Matias, Patrick Mayweg, Gareth McCaughan, Jon Middleton, Tim Moloney, Christopher Ness, Mats Nilsson, Joe Orton, Amy Lyn Pilato, Kevin Pilch-Bisson, Dmitriy Popkov, Michael Price, Mark Proctor, Steffen Prohaska, Daniel Rall, Jack Repenning, Tobias Ringstrom, Garrett Rooney, Joel Rosdahl, Christian Sauer, Larry Shatzer, Russell Steicke, Sander Striker, Erik Sjoelund, Johan Sundstroem, John Szakmeister, Mason Thomas, Eric Wadsworth, Colin Watson, Alex Waugh, Chad Whitacre, Josef Wolf, Blair Zajac e a comunidade inteira do Subversion.

Agradecimentos de Ben Collins-Sussman

Agradeço a minha esposa Frances, quem, por vários meses, começou a ouvir, “Mas docinho, eu estou trabalhando no livro”, ao invés do habitual, “Mas docinho, eu estou escrevendo um e-mail.” Eu não sei onde ela arruma tanta paciência! Ela é meu equilíbrio perfeito.

Agradeço à minha extensa família e amigos por seus sinceros votos de encorajamento, apesar de não terem qualquer interesse no assunto. (Você sabe, tem uns que dizem: “Você escreveu um livro?”, e então quando você diz que é um livro de computador, eles te olham torto.)

Agradeço aos meus amigos mais próximos, que me fazem um rico, rico homem. Não me olhem assim —vocês sabem quem são.

Agradeço aos meus pais pela minha perfeita formação básica, e pelos inacreditáveis conselhos. Agradeço ao meu filho pela oportunidade de passar isto adiante.

Agradecimentos de Brian W. Fitzpatrick

Um grande obrigado à minha esposa Marie por sua inacreditável compreensão, apoio, e acima de tudo, paciência. Obrigado ao meu irmão Eric, quem primeiro me apresentou à programação voltada para UNIX. Agradeço à minha mãe e minha avó por seu apoio, sem falar nas longas férias de Natal, quando eu chegava em casa e mergulhava minha cabeça no laptop para trabalhar no livro.

Mike e Ben, foi um prazer trabalhar com vocês neste livro. Heck, é um prazer trabalhar com você nesta obra!

Para todos da comunidade Subversion e a Apache Software Foundation, agradeço por me receberem. Não há um dia onde eu não aprenda algo com pelo menos um de vocês.

Finalmente, agradeço ao meu avô que sempre me disse “Liberdade é igual responsabilidade.” Eu não poderia estar mais de acordo.

Agradecimentos de C. Michael Pilato

Um obrigado especial a Amy, minha melhor amiga e esposa por inacreditáveis nove anos, por seu amor e apoio paciente, por me tolerar até tarde da noite, e por agüentar o duro processo de controle de versão que impus a ela. Não se preocupe, querida—você será um assistente do TortoiseSVN logo!

[1]Oh, e agradecemos ao Karl, por ter dedicado muito trabalho ao escrever este livro sozinho.

Gavin, provavelmente não há muitas palavras neste livro que você possa, com sucesso, “pronunciar” nesta fase de sua vida, mas quando você, finalmente, aprender a forma escrita desta louca língua que falamos, espero que você esteja tão orgulhoso de seu pai quanto ele de você.

Aidan, Daddy luffoo et ope Aiduh yike contootoo as much as Aiduh yike batetball, base-ball, et bootball. [2]

Mãe e Pai, agraço pelo apoio e entusiasmo constante. Sogra e Sogro, agradeço por tudo da mesma forma e *mais* um pouco por sua fabulosa filha.

Tiro o chapéu para Shep Kendall, foi através dele que o mundo dos computadores foi aberto pela primeira vez a mim; Ben Collins-Sussman, meu orientador pelo mundo do código-aberto; Karl Fogel — você *é* meu `.emacs`; Greg Stein, o difusor da programação prática como-fazer; Brian Fitzpatrick— por compartilhar esta experiência de escrever junto comigo. Às muitas pessoas com as quais eu estou constantemente aprendendo—e continuo aprendendo!

Finalmente, agradeço a alguém que demonstra ser perfeitamente criativo em sua excelência—você.

O Que é o Subversion?

Subversion é um sistema de controle de versão livre/open-source. Isto é, o Subversion gerencia arquivos e diretórios, e as modificações feitas neles ao longo do tempo. Isto permite que você recupere versões antigas de seus dados, ou que examine o histórico de suas alterações. Devido a isso, muitas pessoas tratam um sistema de controle de versão como uma espécie de “máquina do tempo”.

O Subversion pode funcionar em rede, o que lhe possibilita ser usado por pessoas em diferentes computadores. Em certo nível, a capacidade de várias pessoas modificarem e gerenciarem o mesmo conjunto de dados de seus próprios locais é o que fomenta a colaboração. Progressos podem ocorrer muito mais rapidamente quando não há um gargalo único por onde todas as modificações devam acontecer. E como o trabalho está versionado, você não precisa ter medo de que seu trabalho perca qualidade por não ter essa via única para modificações—se os dados sofrerem alguma modificação indevida, apenas desfaça tal modificação.

Alguns sistemas de controle de versão também são sistema de gerenciamento de configuração (GC). Estes sistemas são especificamente desenvolvimento para gerenciar árvores de código-fonte, e possuem muitos recursos específicos para o desenvolvimento de software—como identificação nativa de linguagens de programação, ou ferramentas de apoio para compilação de software. O Subversion, no entanto, não é um sistema desse tipo. É um sistema de caráter geral que pode ser usado para gerenciar *quaisquer* conjuntos de arquivos. Para você, estes arquivos podem ser código-fonte—para outros, podem ser qualquer coisa desde listas de compras de supermercado a arquivos de edição de vídeo, e muito mais.

Histórico do Subversion

No começo do ano 2000, a CollabNet, Inc. (http://www.collab.net) começou a procurar desenvolvedores para desenvolver um substituto para o CVS. A CollabNet já tinha uma suite colaborativa chamada CollabNet Enterprise Edition (CEE) cujo um de seus componentes era o controle de versão. Apesar de o CEE usar o CVS como seu sistema de controle de versão inicial, as limitações do CVS ficaram evidentes desde o princípio, e a CollabNet sabia que eventualmente teria que procurar por algo melhor. Infelizmente, o CVS havia se firmado como um padrão de fact no mundo open source principalmente porque *não havia* nada melhor, pelo menos sob licença livre. Então a CollabNet decidiu desenvolver um novo sistema de controle de versão a partir do zero, mantendo as idéias básicas do CVS, mas sem os bugs e seus inconvenientes.

Em Fevereiro de 2000, eles contactaram Karl Fogel, o autor de *Open Source Development with CVS* (Coriolis, 1999), e perguntaram se ele gostaria de trabalhar neste novo projeto. Coincidentemente,

[2]Tradução: Papai te ama e espera que você goste de computadores assim como você irá gostar de basquete, basebol e futebol. (Isso seria óbvio?)

no momento Karl já estava discutindo o projeto para um novo sistema de controle de versão com seu amigo Jim Blandy. Em 1995, os dois iniciaram a Cyclic Software, uma empresa que mantinha contratos de suporte para o CVS, e apesar de terem vendido a empresa posteriormente, eles ainda usavam o CVS todos os dias em seus trabalhos. Suas frustrações com o CVS levou Jim a pensar cuidadosamente sobre melhores maneiras para gerenciar dados versionados, no que ele não apenas já tinha pensado no nome “Subversion”, mas também com o projeto básico para armazenamento de dados do Subversion. Quando a CollabNet chamou, Karl concordou imediatamente em trabalhar no projeto, e Jim sugeriu à empresa em que trabalhava, Red Hat Software, essencialmente a cedê-lo para o projeto por um período de tempo indefinido. A CollabNet contratou Karl e Ben Collins-Sussman, e um projeto detalhado de trabalho começou em Maio. Com a ajuda e o bem-vindo incentivo de Brian Behlendorf e Jason Robbins da CollabNet, e de Greg Stein (à época, um desenvolvedor independente trabalhando no processo de especificação do WebDAV/DeltaV), o Subversion rapidamente atraiu uma comunidade ativa de desenvolvedores. Detectou-se que muitas pessoas também tinham as mesmas experiências frustrantes com o CVS, agora satisfeitas com a oportunidade de finalmente fazer algo sobre isso.

A equipe do projeto original determinou alguns objetivos simples. Eles não queriam romper com a metodologia existente para controle de versão, eles apenas queriam corrigir o CVS. Eles decidiram que o Subversion deveria ser compatível com as características do CVS, e manter o mesmo modelo de desenvolvimento, mas não reproduzir as falhas mais óbvias do CVS. E mesmo que o novo sistema não fosse um substituto definitivo para o CVS, ele deveria ser suficientemente semelhante a este para que qualquer usuário do CVS pudesse migrar de sistema com pouco esforço.

Depois de quatorze meses de desenvolvimento, o Subversion tornou-se “auto-gerenciável” em 31 de Agosto de 2001. Ou seja, os desenvolvedores do Subversion pararam de usar o CVS para gerir seu próprio código-fonte, e começaram a usar o próprio Subversion no lugar.

Embora a CollabNet tenha iniciado o projeto, e ainda patrocine uma grande parte dos trabalhos (ela paga os salários de alguns poucos desenvolvedores do Subversion em tempo integral), o Subversion é mantido como a maioria dos projetos open source, gerenciado por um conjunto de regras transparentes e de senso-comum, fundamentadas na meritocracia. A licença adotada pela CollabNet é perfeitamente compatível com Definição Debian de Software Livre (DFSG). Em outras palavras, qualquer pessoa é livre para baixar o código do Subversion, modificá-lo, e redistribuí-lo conforme lhe convier; não é necessário pedir permissão à CollabNet ou a quem quer que seja.

Características do Subversion

Ao discutir sobre que recursos o Subversion traz para o âmbito do controle de versão, frequentemente é útil falar deles em termos de que avanços eles representam aos recursos do CVS. Se você não está familiarizado com o CVS, você pode não compreender todas essas características. E se você não estiver familiarizado com controle de versão como um todo, você pode ficar bem confuso a menos que você leia antes Capítulo 1, *Conceitos Fundamentais*, onde apresentamos uma suave introdução ao controle de versão.

O Subversion proporciona:

Versionamento de diretórios
: O CVS apenas rastreia o histórico de arquivos individuais, já o Subversion implementa um sistema de arquivos “virtual” sob controle de versão que rastreia modificações a toda a árvore de diretório ao longo do tempo. Os arquivos *e* os diretórios são versionados.

Histórico de versões efetivo
: Como o CVS é limitado apenas ao versionamento de arquivos, operações como cópia e renomeação—que podem ocorrer com arquivos também, mas que são realmente alterações no conteúdo de algum diretório continente—não são suportadas no CVS. Adicionalmente, no CVS você não pode substituir um arquivo versionado por alguma outra coisa com o mesmo nome sem que o novo item deixe de herdar o histórico do arquivo antigo—que talvez seja até algo com o qual não mantenha nenhuma correlação. Com o Subversion, você pode adicionar, excluir, copiar,

e renomear ambos os arquivos ou diretórios. E cada novo arquivo adicionado começa com um histórico próprio e completamente novo.

Commits atômicos
Um conjunto de modificações ou é inteiramente registrado no repositório, ou não é registrado de forma nenhuma. Isto possibilita aos desenvolvedores criarem e registrarem alterações como blocos lógicos, e também evita problemas que possam ocorrer quando apenas uma parte de um conjunto de alterações seja enviada com sucesso ao repositório.

Versionamento de metadados
Cada arquivo e diretório tem um conjunto de propriedades—chaves e seus valores—associados consigo. Você pode criar e armazenar quaisquer pares chave/valor que quiser. As propriedades são versionadas ao longo do tempo, tal como os conteúdos de arquivo.

Escolha das camadas de rede
O Subversion tem uma noção abstrata do acesso ao repositório, tornando-o mais fácil para as pessoas implementarem novos mecanismos de rede. O Subversion pode se associar ao servidor Apache HTTP como um módulo de extensão. Isto dá ao Subversion uma grande vantagem em estabilidade e interoperabilidade, além de acesso instantâneo aos recursos existentes oferecidos por este servidor—autenticação, autorização, compactação online, dentre outros. Um servidor Subversion mais leve e independente também está disponível. Este servidor utiliza um protocolo específico o qual pode ser facilmente ser tunelado sobre SSH.

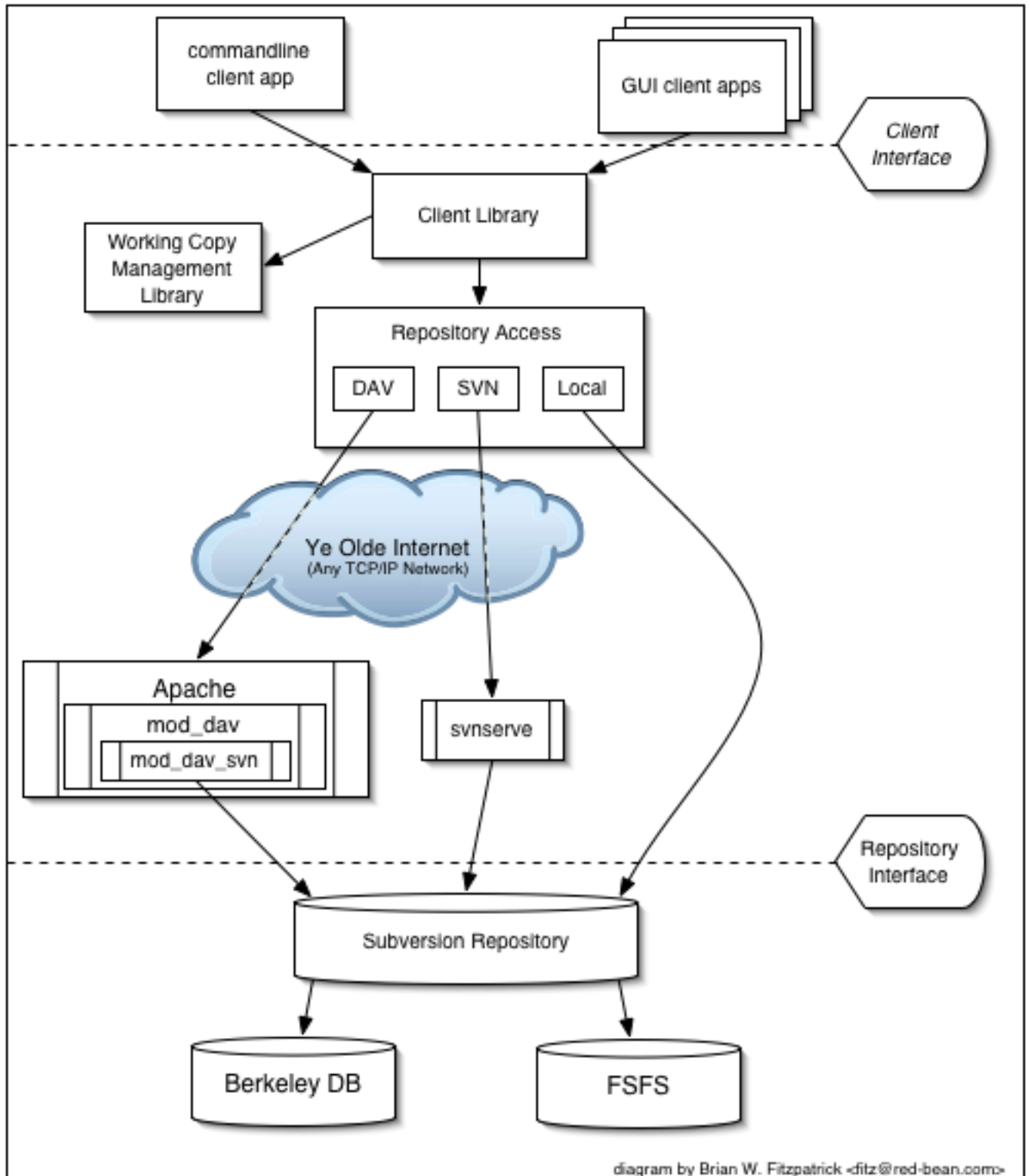
Manipulação consistente de dados
O Subversion exprime as diferenças de arquivo usando um algoritmo diferenciado, o qual funciona de maneira idêntica tanto em arquivos texto (compreensível para humanos) quanto em arquivos binários (incompreensível para humanos). Ambos os tipos de arquivos são igualmente armazenados de forma compactada no repositório, e as diferenças são enviadas em ambas as direções pela rede.

Ramificações e rotulagem eficiente
O custo de se fazer ramificações (*branching*) e de rotulagem (*tagging*) não precisa ser proporcional ao tamanho do projeto. O Subversion cria ramos e rótulos simplesmente copiando o projeto, usando um mecanismo semelhante a um hard-link. Assim essas operações levam apenas uma pequena e constante quantidade de tempo.

Hackability
O Subversion não tem qualquer bagagem histórica; ele é implementado como um conjunto de bibliotecas C compartilhadas com APIs bem definidas. Isto torna o Subversion extremamente manutenível e usável por outras aplicações e linguagens.

Arquitetura do Subversion

Figura 1, “Arquitetura do Subversion” ilustra uma visão em “alto nível” da estrutura do Subversion.

Figura 1. Arquitetura do Subversion

Em uma ponta encontra-se um repositório do Subversion que mantém todos os seus dados versionados. No outro extremo está o seu programa cliente Subversion, que gerencia cópias locais de partes desses dados versionados (chamadas de “cópias de trabalho”). Entre esses dois extremos estão múltiplas rotas por entre várias camadas de Acesso ao Repositório (RA). Algumas dessas rotas partem das redes de computadores até os servidores de rede, de onde então acessam o repositório. Outras desconsideram a rede completamente e acessam diretamente o repositório.

Componentes do Subversion

Uma vez instalado, o Subversion consiste num conjunto de diversas partes. Uma breve visão geral sobre tudo o que você dispõe é mostrada a seguir. Não se preocupe se as breves descrições acabarem fundindo a sua cuca—há *muito* mais páginas neste livro para acabar com essa confusão.

svn
: O programa cliente de linha de comando.

svnversion
: Um programa para informar o estado (em termos das revisões dos itens presentes) da cópia de trabalho.

svnlook
: Uma ferramenta para inspecionar um repositório Subversion diretamente.

svnadmin
: Uma ferramenta para criação, ajuste e manutenção de um repositório Subversion.

svndumpfilter
: Um programa para filtragem de fluxos de um repositório Subversion.

mod_dav_svn
: Um módulo plugin para o servidor Apache HTTP, usado para disponibilizar seu repositório a outros através da rede.

svnserve
: Um específico programa servidor independente, executável como um processo daemon ou invocável via SSH; uma outra forma de disponibilizar seu repositório a outros através da rede.

svnsync
: Um programa para fazer espelhamento incremental de um repositório para outro através da rede.

Uma vez que você tenha instalado o Subversion corretamente, você já deve estar pronto para iniciar. Os próximos dois capítulos vão guiá-lo pela uso do **svn**, o programa cliente de linha de comando do Subversion.

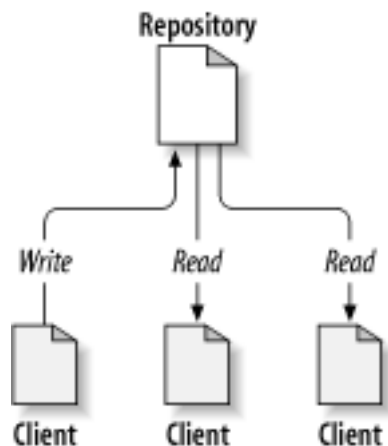
Capítulo 1. Conceitos Fundamentais

Este capítulo é uma breve e casual introdução ao Subversion. Se você é novo em controle de versão, este capítulo é definitivamente para você. Nós começaremos com uma discussão sobre os conceitos gerais de controle de versão, avançaremos para as idéias específicas por trás do Subversion, e mostraremos alguns exemplos simples do Subversion em uso.

Embora os exemplos neste capítulo mostrem pessoas compartilhando coleções de código fonte de programas, tenha em mente que o Subversion pode gerenciar qualquer tipo de coleção de arquivos - ele não está limitado a ajudar programadores.

O Repositório

O Subversion é um sistema centralizado de compartilhamento de informação. Em seu núcleo está um repositório, que é uma central de armazenamento de dados. O repositório armazena informação em forma de uma *árvore de arquivos* - uma hierarquia típica de arquivos e diretórios. Qualquer número de *clientes* se conecta ao repositório, e então lê ou escreve nestes arquivos. Ao gravar dados, um cliente torna a informação disponível para outros; ao ler os dados, o cliente recebe informação de outros. Figura 1.1, "Um típico sistema cliente/servidor" ilustra isso.

Figura 1.1. Um típico sistema cliente/servidor

Então, por que razão isto é interessante? Até ao momento, isto soa como a definição de um típico servidor de arquivos. E, na verdade, o repositório *é* uma espécie de servidor de arquivos, mas não de um tipo comum. O que torna o repositório do Subversion especial é que *ele se lembra de cada alteração* já ocorrida nele: de cada mudança em cada arquivo, e até mesmo alterações na árvore de diretórios em si, como a adição, eliminação, e reorganização de arquivos e diretórios.

Quando um cliente lê dados de um repositório, ele normalmente vê apenas a última versão da árvore de arquivos. Mas o cliente também tem a habilidade de ver os estados *anteriores* do sistema de arquivos. Por exemplo, um cliente pode perguntar questões de histórico como, "O que este diretório continha na última quarta-feira?" ou "Quem foi a última pessoa que alterou este arquivo, e que alterações ela fez?" Estes são os tipos de questões que estão no coração de qualquer *sistema de controle de versão*: sistemas que são projetados para monitorar alterações nos dados ao longo do tempo.

Modelos de Versionamento

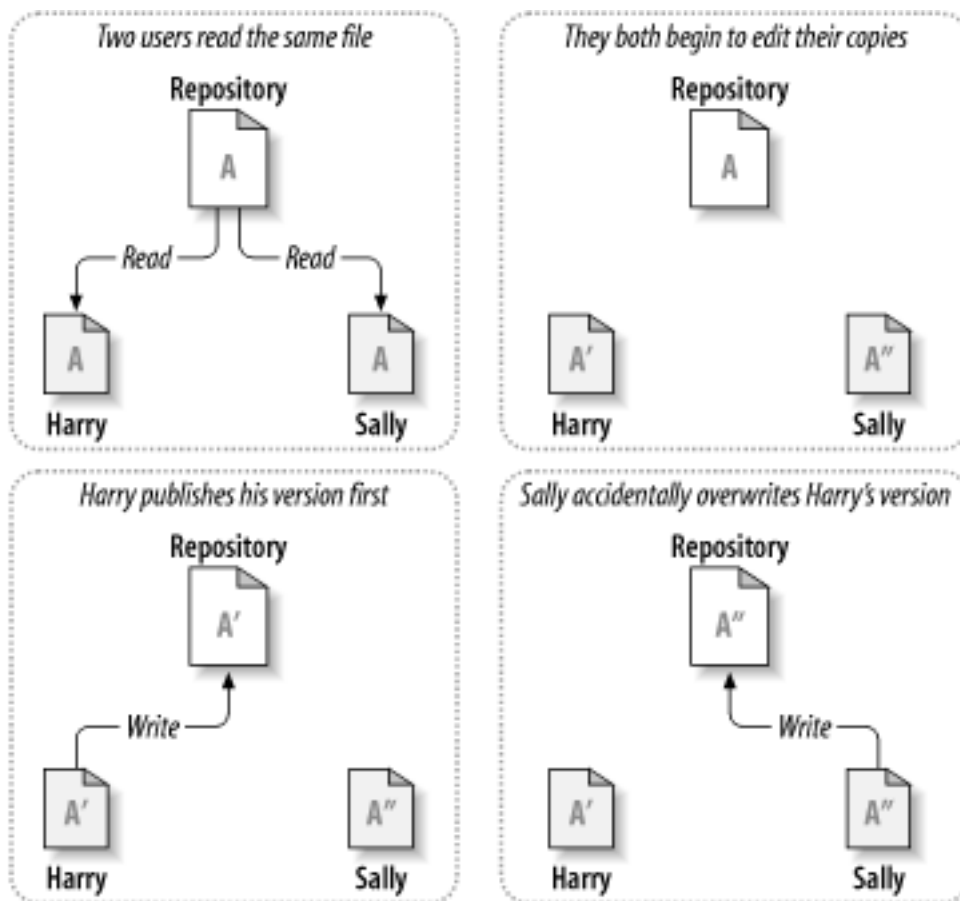
A missão principal de um sistema de controle de versão é permitir a edição colaborativa e o compartilhamento de dados. Mas diferentes sistemas usam diferentes estratégias para atingir esse objetivo. É importante compreender essas diferentes estratégias por várias razões. Primeiro, irá ajudá-lo a comparar os sistemas de controle de versão existentes, no caso de você encontrar outros sistemas

similares ao Subversion. Além disso, irá ajudá-lo ainda a tornar o uso do Subversion mais eficaz, visto que o Subversion por si só permite trabalhar de diferentes formas.

O Problema do Compartilhamento de Arquivos

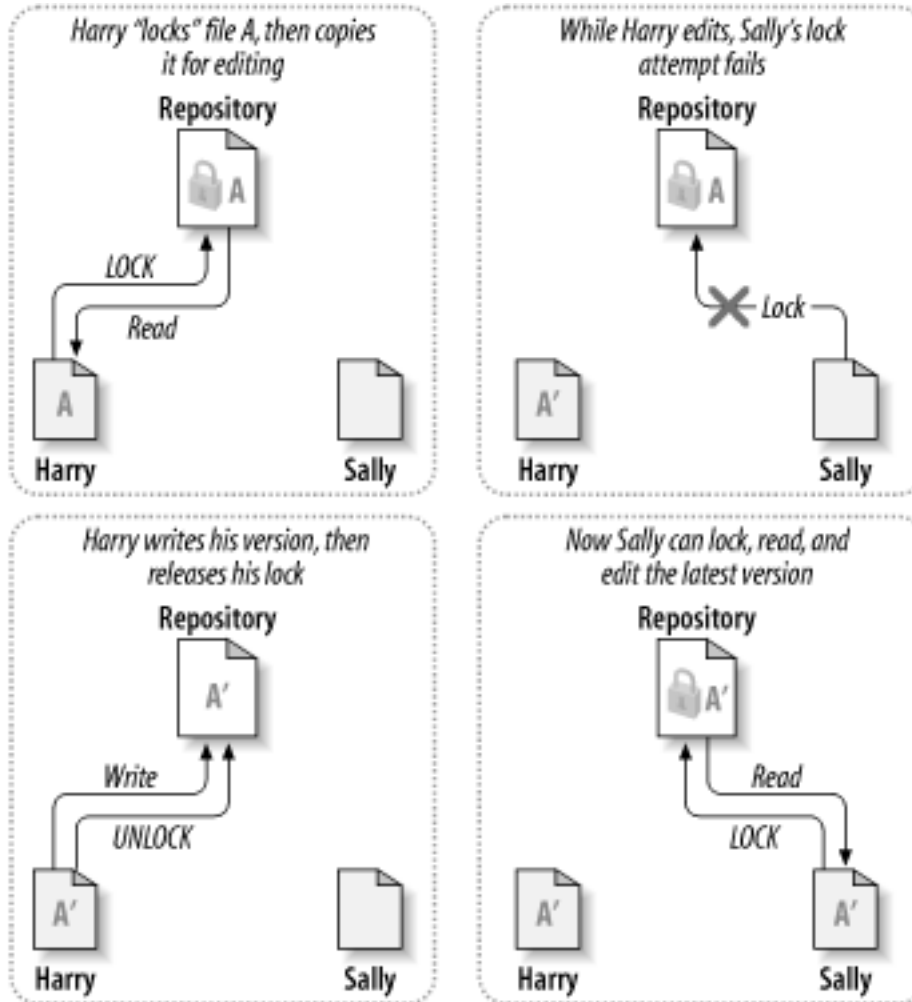
Todos os sistemas de controle de versão têm de resolver o mesmo problema fundamental: como o sistema irá permitir que os usuários compartilhem informação, e como ele irá prevenir que eles acidentalmente tropecem uns nos pés dos outros? É muito fácil para os usuários acidentalmente sobrescrever as mudanças feitas pelos outros no repositório.

Considere o cenário mostrado em Figura 1.2, “O problema para evitar”. Vamos supor que nós temos dois colegas de trabalho, Harry e Sally. Cada um deles decide editar o mesmo arquivo no repositório ao mesmo tempo. Se Harry salvar suas alterações no repositório primeiro, então é possível que (poucos momentos depois) Sally possa acidentalmente sobrescrevê-lo com a sua própria nova versão do arquivo. Embora a versão de Harry não seja perdida para sempre (porque o sistema se lembra de cada mudança), todas as mudanças feitas por Harry *não* vão estar presentes na versão mais recente do arquivo de Sally, porque ela nunca viu as mudanças de Harry's para começar. O trabalho de Harry efetivamente se perdeu - ou pelo menos desapareceu da última versão do arquivo - e provavelmente por acidente. Trata-se definitivamente de uma situação que queremos evitar!

Figura 1.2. O problema para evitar

A Solução Lock-Modify-Unlock

Muitos sistemas de controle de versão usam o modelo *lock-modify-unlock* (travar-modificar-destravar) para resolver o problema de vários autores destruírem o trabalho uns dos outros. Neste modelo, o repositório permite que apenas uma pessoa de cada vez altere o arquivo. Essa política de exclusividade é gerenciada usando locks (travas). Harry precisa “travar ” (lock) um arquivo antes que possa fazer alterações nele. Se Harry tiver travado o arquivo, então Sally não poderá travá-lo também, e portanto,

não poderá fazer nenhuma alteração nele. Tudo que ela pode fazer é ler o arquivo, e esperar que Harry termine suas alterações e destrave (unlock) o arquivo. Depois que Harry destravar o arquivo, Sally poderá ter a sua chance de travar e editar o arquivo. A figura Figura 1.3, "A solução lock-modify-unlock" demonstra essa solução simples.

Figura 1.3. A solução lock-modify-unlock

O problema com o modelo lock-modify-unlock é que ele é um pouco restritivo, muitas vezes se torna um obstáculo para os usuários:

- *Locks podem causar problemas administrativos.* Algumas vezes Harry irá travar o arquivo e se esquecer disso. Entretanto, devido a Sally ainda estar esperando para editar o arquivo, suas mãos estão atadas. E Harry então sai de férias. Agora Sally tem que pedir a um administrador para destravar o arquivo que Harry travou. Essa situação acaba causando uma série de atrasos desnecessários e perda de tempo.

- *Locking pode causar serialização desnecessária.* E se Harry está editando o começo de um arquivo de texto, e Sally simplesmente quer editar o final do mesmo arquivo? Essas mudanças não vão se sobrepor afinal. Eles podem facilmente editar o arquivo simultaneamente, sem grandes danos, assumindo que as alterações serão apropriadamente fundidas depois. Não há necessidade de se trabalhar em turnos nessa situação.

- *Locking pode criar falsa sensação de segurança.* Suponha que Harry trave e edite o arquivo A, enquanto Sally simultaneamente trava e edita o arquivo B. Mas e se A e B dependem um do outro, e se as mudanças feitas em cada são semanticamente incompatíveis? Subitamente A e B não funcionam juntos mais. O sistema de locking não foi suficientemente poderoso para prevenir o

problema - ainda que de certa forma tenha proporcionado uma falsa sensação de segurança. É fácil para Harry e Sally imaginar que travando os arquivos, cada um está começando uma tarefa isolada segura, e assim não se preocupar em discutir as incompatibilidades que virão com suas mudanças. Locking freqüentemente se torna um substituto para a comunicação real.

A Solução Copy-Modify-Merge

O Subversion, CVS, e muitos outros sistemas de controle de versão usam um modelo de *copy-modify-merge* (copiar-modificar-fundir) como uma alternativa ao locking. Nesse modelo, cada usuário se conecta ao repositório do projeto e cria uma *cópia de trabalho* pessoal (personal working copy, ou cópia local) - um espelho local dos arquivos e diretórios do repositório. Os usuários então trabalham simultaneamente e independentemente, modificando suas cópias privadas. Finalmente, as cópias privadas são fundidas (merged) numa nova versão final. O sistema de controle de versão freqüentemente ajuda com a fusão, mas, no final, a intervenção humana é a única capaz de garantir que as mudanças foram realizadas de forma correta.

Aqui vai um exemplo. Digamos que Harry e Sally criaram cada um a sua cópia de trabalho de um mesmo projeto, copiadas do repositório. Eles trabalharam simultaneamente fazendo alterações no arquivo A nas suas próprias cópias. Sally salva suas alterações no repositório primeiro. Quando Harry tentar salvar suas alterações mais tarde, o repositório vai informá-lo que seu arquivo A está *desatualizado* (out-of-date). Em outras palavras, o arquivo A do repositório foi de alguma forma alterado desde a última vez que ele foi copiado. Então Harry pede a seu programa cliente para ajudá-lo a *fundir* (merge) todas as alterações do repositório na sua cópia de trabalho do arquivo A. Provavelmente, as mudanças de Sally não se sobrepõem com as suas próprias; então, uma vez que ele tiver ambos os conjuntos de alterações integrados, ele salva sua cópia de trabalho de volta no repositório. As figuras Figura 1.4, “A solução copy-modify-merge” e Figura 1.5, “A solução copy-modify-merge (continuando)” mostram este processo.

Figura 1.4. A solução copy-modify-merge

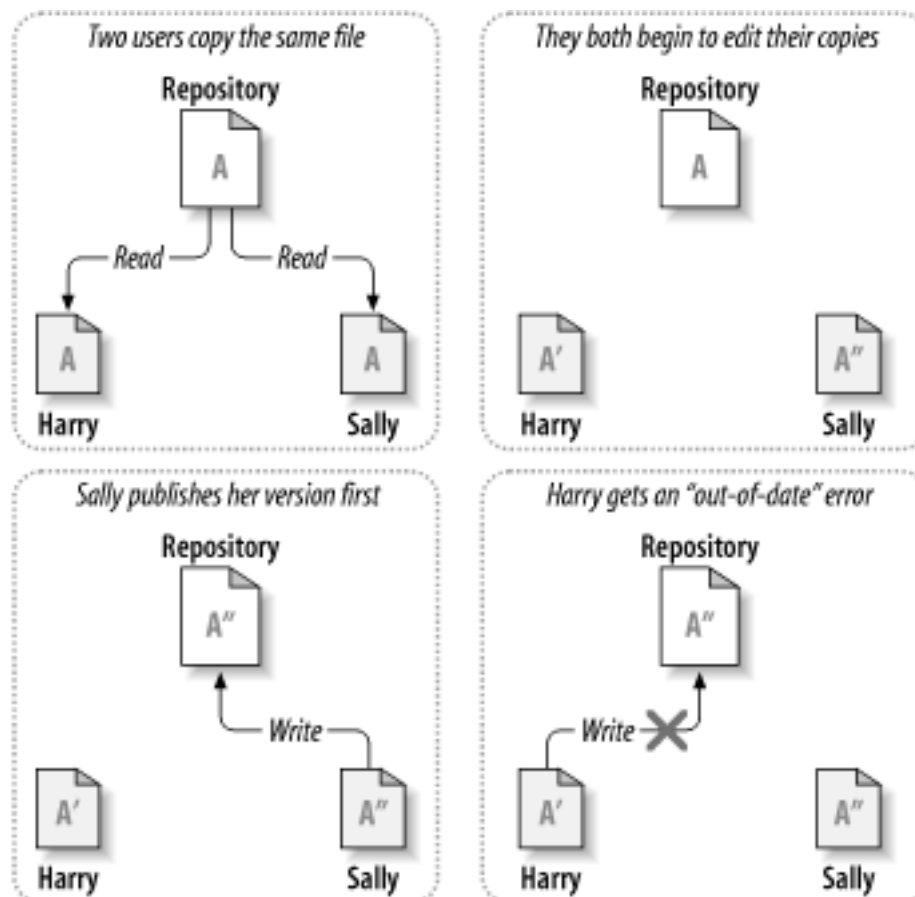
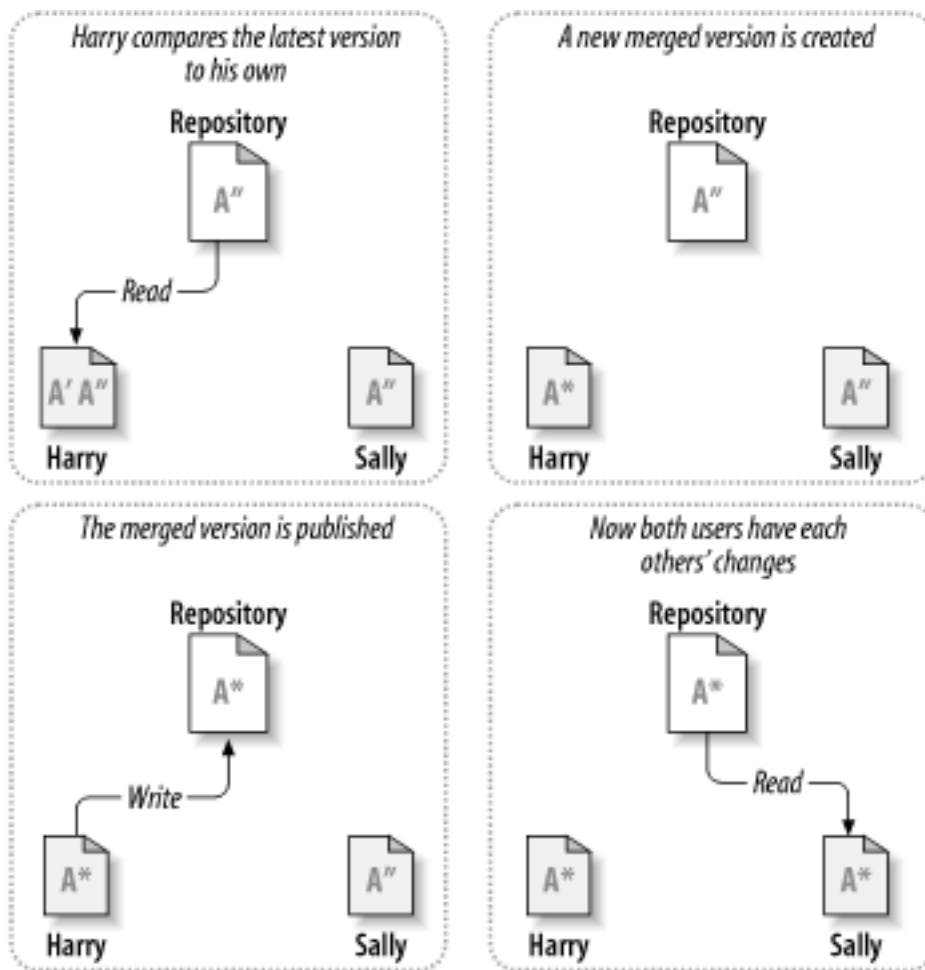



Figura 1.5. A solução copy-modify-merge (continuando)

Mas e se as alterações de Sally *sobrescreverem* as de Harry? E então? Essa situação é chamada de *conflito* , e normalmente não é um problema. Quando Harry pedir a seu cliente para fundir as últimas alterações do repositório em sua cópia de trabalho local, sua cópia do arquivo A estará de alguma forma sinalizada como estando numa situação de conflito: ele será capaz de ver ambos os conjuntos de alterações conflitantes e manualmente escolher entre elas. Note que o software não tem como resolver os conflitos automaticamente; apenas pessoas são capazes de compreender e fazer as escolhas inteligentes. Uma vez que Harry tenha resolvido manualmente as alterações conflitantes - talvez depois de uma conversa com Sally - ele poderá tranqüilamente salvar o arquivo fundido no repositório.

O modelo copy-modify-merge pode soar um pouco caótico, mas, na prática, ele funciona de forma bastante suave. Os usuários podem trabalhar em paralelo, nunca esperando uns pelos outros. Quando eles trabalham nos mesmos arquivos, verifica-se que a maioria de suas alterações simultâneas não se sobrepõe afinal; conflitos não são muito freqüentes. E a quantidade de tempo que eles levam para resolver os conflitos é geralmente muito menor que o tempo perdido no sistema de locking.

No fim, tudo se reduz a um fator crítico: a comunicação entre os usuários. Quando os usuários se comunicam mal, tanto conflitos sintáticos como semânticos aumentam. Nenhum sistema pode forçar os usuários a se comunicarem perfeitamente, e nenhum sistema pode detectar conflitos semânticos. Portanto, não há como confiar nessa falsa sensação de segurança de que o sistema de locking vai prevenir conflitos; na prática, o lock parece inibir a produtividade mais do que qualquer outra coisa.

Quando Lock é Necessário

Enquanto o modelo lock-modify-unlock é geralmente considerado prejudicial à colaboração, ainda há momentos em que ele é apropriado.

O modelo copy-modify-merge é baseado no pressuposto de que os arquivos são contextualmente fundíveis: isto é, que os arquivos no repositório sejam majoritariamente texto plano (como código-fonte). Mas para arquivos com formatos binários, como os de imagens ou som, freqüentemente é impossível fundir as mudanças conflitantes. Nessas situações, é realmente necessário que o arquivo seja alterado por um usuário de cada vez. Sem um acesso serializado, alguém acabará perdendo tempo em mudanças que no final serão descartadas.

Enquanto o Subversion é primariamente um sistema copy-modify-merge, ele ainda reconhece a necessidade ocasional de locking em algum arquivo e assim fornece mecanismos para isso. Este recurso será discutido mais tarde neste livro, em “Travamento”.

Subversion em Ação

Chegou a hora de passar do abstrato para o concreto. Nesta seção, nós mostraremos exemplos reais de utilização do Subversion

URLs do Repositório Subversion

Ao longo de todo este livro, o Subversion utiliza URLs para identificar arquivos e diretórios versionados nos repositórios. Na maior parte, essas URLs usam a sintaxe padrão, permitindo nomes de servidor e números de porta serem especificados como parte da URL:

```
$ svn checkout http://svn.example.com:9834/repos
...
```

Mas existem algumas nuances no manuseio de URLs pelo Subversion que são notáveis. Por exemplo, URLs contendo o método de acesso `file://` (usado para repositórios locais) precisam, de acordo com a convenção, ter como nome do servidor `localhost` ou nenhum nome de servidor:

```
$ svn checkout file:///path/to/repos
...
$ svn checkout file://localhost/path/to/repos
...
```

Além disso, usuários do esquema `file://` em plataformas Windows precisarão utilizar um padrão de sintaxe “não-oficial” para acessar repositórios que estão na mesma máquina, mas em um drive diferente do atual drive de trabalho. Qualquer uma das seguintes sintaxes de URLs funcionarão, sendo X o drive onde o repositório reside:

```
C:\> svn checkout file:///X:/path/to/repos
...
C:\> svn checkout "file:///X|/path/to/repos"
...
```

Na segunda sintaxe, você precisa colocar a URL entre aspas de modo que o caractere de barra vertical não seja interpretado como um pipe. Além disso, note que a URL utiliza barras normais, enquanto no Windows os caminhos (não URLs) utilizam barra invertida.

URLs `file://` do Subversion não podem ser utilizadas em um browser comum da mesma forma que URLs `file://` típicas podem. Quando você tenta ver uma URL `file://` num

web browser comum, ele lê e mostra o conteúdo do local examinando o sistema de arquivos diretamente. Entretanto, os recursos do Subversion existem em um sistema de arquivos virtual (veja “Camada de Repositório”), e o seu browser não vai saber como interagir com este sistema de arquivos.

Por último, convém notar que o cliente Subversion vai automaticamente codificar as URLs conforme necessário, de forma semelhante a um browser. Por exemplo, se a URL contiver espaços ou algum caractere não-ASCII:

```
$ svn checkout "http://host/path with space/project/españa"
```

...então o Subversion irá aplicar "escape" aos caracteres inseguros e se comportar como se você tivesse digitado:

```
$ svn checkout http://host/path%20with%20space/project/espa%C3%B1a
```

Se a URL contiver espaços, certifique-se de colocá-la entre aspas, de forma que o seu shell trate-a inteiramente como um único argumento do programa **svn**.

Cópias de Trabalho, ou Cópias Locais

Você já leu sobre as cópias de trabalho; agora vamos demonstrar como o cliente do Subversion as cria e usa.

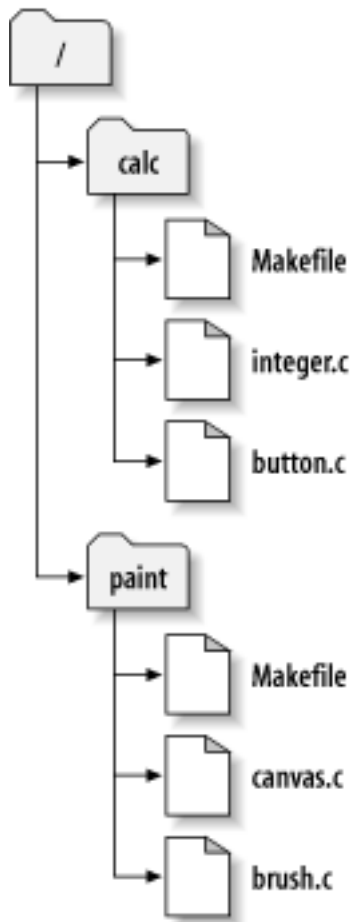
Uma cópia de trabalho do Subversion é uma árvore de diretórios comum no seu sistema de arquivos local, contendo uma coleção de arquivos. Você pode editar esses arquivos conforme desejar, e se eles são arquivos de código fonte, você pode compilar o seu programa a partir deles da maneira usual. Sua cópia de local é sua área de trabalho privada: O Subversion jamais incorporará as mudanças de terceiros ou tornará as suas próprias alterações disponíveis para os outros, até que você explicitamente o diga para fazer isso. Você pode ter múltiplas cópias de trabalho do o mesmo projeto.

Após você ter feito algumas alterações nos arquivos de sua cópia de trabalho e verificado que elas funcionam corretamente, o Subversion lhe disponibiliza comandos para “publicar” (commit) suas alterações para as outras pessoas que estão trabalhando com você no mesmo projeto (gravando no repositório). Se outras pessoas publicarem alterações, o Subversion disponibiliza comandos para fundir (merge) essas alterações em sua cópia de trabalho (lendo do repositório).

Uma cópia de trabalho também contém alguns arquivos extras, criados e mantidos pelo Subversion, para ajudá-lo a executar esse comandos. Em particular, cada diretório em sua cópia local contém um subdiretório chamado `.svn`, também conhecido como o *diretório administrativo* da cópia de local. Os arquivos em cada diretório administrativo ajudam o Subversion a reconhecer quais arquivos possuem alterações não-publicadas, e quais estão desatualizados em relação ao trabalho dos outros.

Um típico repositório Subversion freqüentemente detém os arquivos (ou código fonte) para vários projetos, geralmente, cada projeto é um subdiretório na árvore de arquivos do repositório. Desse modo, uma cópia de trabalho de um normalmente corresponderá a uma sub-árvore particular do repositório.

Por exemplo, suponha que você tenha um repositório que contenha dois projetos de software, `paint` e `calc`. Cada projeto reside em seu próprio subdiretório, como é mostrado em Figura 1.6, “O Sistema de Arquivos do Repositório”.

Figura 1.6. O Sistema de Arquivos do Repositório

Para obter uma cópia local, você deve fazer *check out* de alguma sub-árvore do repositório. (O termo “check out” pode soar como algo que tem a ver com locking ou com reserva de recursos, o que não é verdade; ele simplesmente cria uma cópia privada do projeto para você.) Por exemplo, se você fizer check out de `/calc`, você receberá uma cópia de trabalho como esta:

```
$ svn checkout http://svn.example.com/repos/calc
A    calc/Makefile
A    calc/integer.c
A    calc/button.c
Checked out revision 56.

$ ls -A calc
Makefile  integer.c  button.c  .svn/
```

A lista de letras A na margem esquerda indica que o Subversion está adicionando um certo número de itens à sua cópia de trabalho. Você tem agora uma cópia pessoal do diretório `/calc` do repositório, com uma entrada adicional - `.svn` - a qual detém as informações extras que o Subversion precisa, conforme mencionado anteriormente.

Suponha que você faça alterações no arquivo `button.c`. Visto que o diretório `.svn` se lembra da data de modificação e conteúdo do arquivo original, o Subversion tem como saber que você modificou o arquivo. Entretanto o Subversion não torna as suas alterações públicas até você explicitamente lhe dizer para fazer isto. O ato de publicar as suas alterações é conhecido como *committing* (ou *checking in*) no repositório.

Para publicar as suas alterações para os outros, você deve usar o comando **commit** do Subversion.

```
$ svn commit button.c -m "Fixed a typo in button.c."
Sending        button.c
Transmitting file data .
Committed revision 57.
```

Agora as suas alterações no arquivo `button.c` foram "submetidas" no repositório, com uma nota descrevendo as suas alterações (especificamente você corrigiu um erro de digitação). Se outros usuários fizerem check out de `/calc`, eles verão suas alterações na última versão do arquivo.

Suponha que você tenha um colaborador, Sally, que tenha feito check out de `/calc` ao mesmo tempo que você. Quando você publicar suas alterações em `button.c`, a cópia de trabalho de Sally será deixada intacta; o Subversion somente modifica as cópias locais quando o usuário requisita.

Para atualizar o seu projeto, Sally pede ao Subversion para realizar um *update* na cópia de trabalho dela, usando o comando **update** do Subversion. Isto irá incorporar as suas alterações na cópia local dela, bem como as alterações de todos que tenham feito um commit desde que ela fez check out.

```
$ pwd
/home/sally/calc

$ ls -A
.svn/ Makefile integer.c button.c

$ svn update
U    button.c
Updated to revision 57.
```

A saída do comando **svn update** indica que o Subversion atualizou o conteúdo de `button.c`. Note que Sally não precisou especificar quais arquivos seriam atualizados; o Subversion usou as informações no diretório `.svn`, e mais algumas no repositório, para decidir quais arquivos precisariam ser atualizados.

URLs do Repositório

Os repositórios do Subversion podem ser acessados através de diversos métodos - em um disco local, através de vários protocolos de rede, dependendo de como o administrador configurou as coisas para você. Qualquer local no repositório, entretanto, é sempre uma URL. A Tabela Tabela 1.1, "URLs de Acesso ao Repositório" descreve como diferentes esquemas de URLs mapeiam para os métodos de acesso disponíveis.

Tabela 1.1. URLs de Acesso ao Repositório

Esquema	Método de Acesso
`file:///`	acesso direto ao repositório (em um disco local).
`http://`	acesso via protocolo WebDAV em um servidor Apache especialmente configurado.
`https://`	mesmo que `http://`, mas com encriptação SSL.
`svn://`	acesso via protocolo próprio em um servidor `svnserve`.
`svn+ssh://`	mesmo que `svn://`, mas através de um túnel SSH.

Para obter mais informações sobre como o Subversion analisa as URLs, veja "URLs do Repositório Subversion". Para obter mais informações sobre os diferentes tipos de servidores de rede disponíveis para Subversion, veja Capítulo 6, *Configuração do Servidor*.

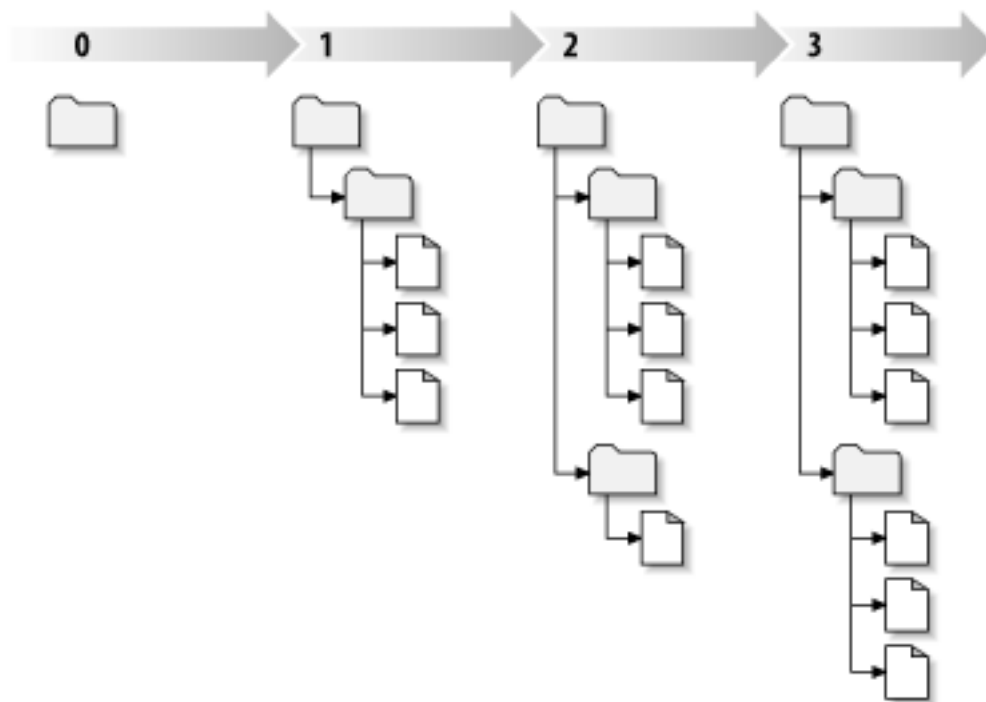
Revisões

Uma operação **svn commit** publica as alterações feitas em qualquer número de arquivos o diretórios como uma única transação atômica. Em sua cópia de trabalho, você pode alterar o conteúdo de arquivos; criar, deletar, renomear e copiar arquivos e diretórios; e então submeter um conjunto completo de alterações em uma transação atômica.

Por “transação atômica”, nos entendemos simplesmente isto: Ou são efetivadas todas as alterações no repositório, ou nenhuma delas. O Subversion tenta manter esta atomicidade em face de quebras ou travamentos do programa ou do sistema, problemas de rede ou outras ações de usuários.

Cada vez que o repositório aceita um commit, isto cria um novo estado na árvore de arquivos, chamado *revisão*. Cada revisão é assinalada com um único número natural, incrementado de um em relação à revisão anterior. A revisão inicial de um repositório recém criado é numerada com zero, e consiste em nada além de um diretório raiz vazio.

A figura Figura 1.7, “O Repositório” ilustra uma forma simples para visualizar o repositório. Imagine um array de números de revisões, iniciando em zero, alongando-se da esquerda para a direita. Cada número de revisão tem uma árvore de arquivos pendurada abaixo dela, e cada árvore é um “snapshot” da forma como o repositório podia ser visto após um commit.

Figura 1.7. O Repositório

> **Números de Revisão Globais**
>
> Ao contrários de outros sistemas de controle de versão, os números de revisão do Subversion se aplicam à *árvore inteira*, não a arquivos individuais. Cada número de revisão refere-se a uma árvore inteira, um estado particular do repositório após determinadas alterações serem submetidas. Uma outra forma de pensar a respeito é imaginar que a revisão N representa o estado do sistema de arquivos do repositório após o N-ésimo commit. Quando os usuários do Subversion falam sobre a “revisão número 5 do arquivo `foo.c`”, eles realmente entendem o “`foo.c` que aparece na revisão 5.” Note que em geral, revisões N e M de um arquivo podem *não* ser necessariamente diferentes! Muitos outros sistemas de controle de versão usam número de revisão por arquivo, então este conceito pode parecer não usual à primeira vista. (Usuários do CVS podem querer ver Apêndice B, *Subversion para Usuários de CVS* para mais detalhes.)

É importante notar que nem sempre as cópias de trabalho correspondem a uma única revisão do repositório; elas podem conter arquivos de várias revisões diferentes. Por exemplo, suponha que você faça checkout de uma cópia de trabalho cuja revisão mais recente seja 4:

```
calc/Makefile:4
     integer.c:4
     button.c:4
```

Neste momento, este diretório de trabalho corresponde exatamente à revisão número 4 no repositório. Contudo, suponha que você faça uma alteração no arquivo `button.c`, e publique essa alteração. Assumindo que nenhum outro commit tenha sido feito, o seu commit irá criar a revisão 5 no repositório, e sua cópia de trabalho agora irá parecer com isto:

```
calc/Makefile:4
     integer.c:4
     button.c:5
```

Suponha que neste ponto, Sally publique uma alteração no arquivo `integer.c`, criando a revisão 6. Se você usar o comando **svn update** para atualizar a sua cópia de trabalho, então ela irá parecer com isto:

```
calc/Makefile:6
     integer.c:6
     button.c:6
```

A alteração de Sally no arquivo `integer.c` irá aparecer em sua cópia de trabalho, e a sua alteração no arquivo `button.c` ainda estará presente. Neste exemplo, o texto do arquivo `Makefile` é idêntico nas revisões 4, 5, e 6, mas o Subversion irá marcar a sua cópia do arquivo `Makefile` com a revisão 6 para indicar que a mesma é a corrente. Então, depois de você fazer uma atualização completa na sua cópia de trabalho, ela geralmente corresponderá exatamente a uma revisão do repositório.

Como as Cópias de Trabalho Acompanham o Repositório

Para cada arquivo em um diretório de trabalho, o Subversion registra duas peças de informações essenciais na área administrativa `.svn/`:

- em qual revisão o seu arquivo local é baseado (isto é chamado de *revisão local* do arquivo), e
- a data e a hora da última vez que a cópia local foi atualizada a partir do repositório.

Dadas estas informações, conversando com o repositório, o Subversion pode dizer em qual dos seguintes quatro estados um arquivo local está:

Não-Modificado, e corrente
: O arquivo não foi modificado no diretório local, e nenhuma alteração foi publicada no repositório desde a revisão corrente. O comando **svn commit** no arquivo não fará nada, e um comando **svn update** também não..

Localmente alterado, e corrente
: O arquivo foi alterado no diretório local, mas nenhuma alteração foi publicada no repositório desde o último update. existem alterações locais que ainda não foram publicadas no repositório, assim o comando **svn commit** no arquivo resultará na publicação dessas alterações, e um comando **svn update** não fará nada.

Não-Modificado, e desatualizado
: O arquivo não foi alterado no diretório local, mas foi alterado no repositório. O arquivo pode ser eventualmente atualizado, para sincronizá-lo com a última revisão pública. O comando **svn commit**

no arquivo não irá fazer nada, mas o comando **svn update** irá trazer as últimas alterações para a sua cópia local.

Localmente Modificado, e desatualizado
O arquivo foi alterado tanto no diretório loca quanto no repositório. O comando **svn commit** no arquivo irá falhar com o erro “out-of-date” (desatualizado). O arquivo deve ser atualizado primeiro; o comando **svn update** vai tentar fundir as alterações do repositório com as locais. Se o Subversion não conseguir completar a fusão de uma forma plausível automaticamente, ele deixará para o usuário resolver o conflito.

Isto pode soar como muito para acompanhar, mas o comando **svn status** mostrará para você o estado de qualquer item em seu diretório local. Para maiores informações sobre este comando, veja “Obtendo uma visão geral de suas alterações”.

Revisões Locais Mistas

Como um princípio geral, o Subversion tenta ser tão flexível quanto possível. Um tipo especial de flexibilidade é a capacidade de ter uma cópia local contendo arquivos e diretórios com uma mistura de diferentes revisões. Infelizmente esta flexibilidade tende a confundir inúmeros novos usuários. Se o exemplo anterior mostrando revisões mistas deixou você perplexo, aqui está um exemplo mostrando tanto a razão pela qual o funcionalidade existe, quanto como fazer para usá-la.

Atualizações e Submissões são Separados

Uma das regras fundamentais do Subversion é que uma ação de “push” não causa um “pull”, e vice versa. Só porque você está pronto para publicar novas alterações no repositório não significa que você está pronto para receber as alterações de outras pessoas. E se você tiver novas alterações em curso, então o comando **svn update** deveria graciosamente fundir as alterações no repositório com as suas próprias, ao invés de forçar você a publicá-las.

O principal efeito colateral dessa regra significa que uma cópia local tem que fazer uma escrituração extra para acompanhar revisões mistas, bem como ser tolerante a misturas. Isso fica mais complicado pelo fato de os diretórios também serem versionados.

Por exemplo, suponha que você tenha uma cópia local inteiramente na revisão 10. Você edita o arquivo `foo.html` e então realiza um comando **svn commit**, o qual cria a revisão 15 no repositório. Após o commit acontecer, muitos novos usuários poderiam esperar que a cópia local estivesse na revisão 15, mas este não é o caso! Qualquer número de alterações poderia ter acontecido no repositório entre as revisões 10 e 15. O cliente nada sabe sobre essas alterações no repositório, pois você ainda não executou o comando **svn update** , e o comando **svn commit** não baixou as novas alterações no repositório. Se por outro lado, o comando **svn commit** *tivesse* feito o download das novas alterações automaticamente, então seria possível que a cópia local inteira estivesse na revisão 15 - mas então nós teríamos quebrado a regra fundamental onde “push” e “pull” permanecem como ações separadas. Portanto a única coisa segura que o cliente Subversion pode fazer é marcar o arquivo - `foo.html` com a revisão 15. O restante da cópia local permanece na revisão 10. Somente executando o comando **svn update** as alterações mais recentes no repositório serão baixadas, o a cópia local inteira será marcada com a revisão 15.

Revisões misturadas são normais

O fato é, *cada vez* que você executar um comando **svn commit**, sua cópia local acabará tendo uma mistura de revisões. As coisas que você acabou de publicar são marcadas com um número de revisão maior que todo o resto. Após várias submissões (sem atualizações entre eles) sua cópia local irá conter uma completa mistura de revisões. Mesmo que você seja a única pessoa utilizando o repositório, você ainda verá este fenômeno. Para analisar a sua mistura de revisões use o comando **svn status --verbose** (veja “Obtendo uma visão geral de suas alterações” para maiores informações.)

Freqüentemente, os novos usuários nem tomam consciência de que suas cópias locais contêm revisões mistas. Isso pode ser confuso, pois muitos comandos no cliente são sensíveis às revisões que eles

estão examinando. Por exemplo, o comando **svn log** é usado para mostrar o histórico de alterações em um arquivo ou diretório (veja “Gerando uma lista de alterações históricas”). Quando o usuário invoca este comando em um objeto da cópia local, ele espera ver o histórico inteiro do objeto. Mas se a revisão local do objeto é muito velha (muitas vezes porque o comando **svn update** não foi executado por um longo tempo), então o histórico da versão *antiga* do objeto é que será mostrado.

Revisões mistas são úteis

Se o seu projeto for suficientemente complexo, você irá descobrir que algumas vezes é interessante forçar um *backdate* (ou, atualizar para uma revisão mais antiga que a que você tem) de partes de sua cópia local para revisões anteriores; você irá aprender como fazer isso em Capítulo 2, *Uso Básico*. Talvez você queira testar uma versão anterior de um submódulo contido em um subdiretório, ou talvez queira descobrir quando um bug apareceu pela primeira vez eu arquivo específico. Este é o aspecto de “máquina do tempo” de um sistema de controle de versão - a funcionalidade que te permite mover qualquer parte de sua cópia local para frente ou para trás na história.

Revisões mistas têm limitações

Apesar de você poder fazer uso de revisões mistas em seu ambiente local, esta flexibilidade tem limitações.

Primeiramente, você não pode publicar a deleção de um arquivo ou diretório que não esteja completamente atualizado. Se uma versão mais nova do item existe no repositório, sua tentativa de deleção será rejeitada, para prevenir que você acidentalmente destrua alterações que você ainda não viu.

Em segundo lugar, você não pode publicar alterações em metadados de diretórios a menos que ele esteja completamente atualizado. Você irá aprender a anexar “propriedades” aos itens em Capítulo 3, *Tópicos Avançados*. Uma revisão em um diretório local define um conjunto específico de entradas e propriedades, e assim, publicar alterações em propriedades de um diretório desatualizado pode destruir propriedades que você ainda não viu.

Sumário

Nós abordamos uma série de conceitos fundamentais do Subversion neste capítulo:

- Nós introduzimos as noções de repositório central, cópia local do cliente, e o array de árvores de revisões.
- Vimos alguns exemplos simples de como dois colaboradores podem utilizar o Subversion para publicar e receber as alterações um do outro, utilizando o modelo “ copy-modify-merge”.
- Nós falamos um pouco sobre a maneira como o Subversion acompanha e gerencia as informações de uma cópia local do repositório.

Neste ponto, você deve ter uma boa idéia de como o Subversion funciona no sentido mais geral. Com este conhecimento, você já deve estar pronto para avançar para o próximo capítulo, que é um relato detalhado dos comandos e recursos do Subversion.

Capítulo 2. Uso Básico

Agora entraremos nos detalhes do uso do Subversion. Quando chegar ao final deste capítulo, você será capaz de realizar todas as tarefas necessárias para usar Subversion em um dia normal de trabalho. Iniciará acessando seus arquivos que estão no Subversion, após ter obtido uma cópia inicial de seu código. Guiaremos você pelo processo de fazer modificações e examinar estas modificações. Também verá como trazer mudanças feitas por outros para sua cópia de trabalho, examiná-las, e resolver quaisquer conflitos que possam surgir.

Note que este capítulo não pretende ser uma lista exaustiva de todos os comandos do Subversion—antes, é uma introdução conversacional às tarefas mais comuns que você encontrará no Subversion. Este capítulo assume que você leu e entendeu o Capítulo 1, *Conceitos Fundamentais* e está familiarizado com o modelo geral do Subversion. Para uma referência completa de todos os comandos, veja Capítulo 9, *Referência Completa do Subversion*.

Help!

Antes de continuar a leitura, aqui está o comando mais importante que você precisará quando usar o Subversion: **svn help**. O cliente de linha de comando do Subversion é auto-documentado—a qualquer momento, um rápido **svn help *`SUBCOMANDO`*** descreverá a sintaxe, opções, e comportamento do subcomando.

```
$ svn help import
import: Faz commit de um arquivo não versionado ou árvore no repositório.
uso: import [CAMINHO] URL

  Recursivamente faz commit de uma cópia de CAMINHO para URL.
  Se CAMINHO é omitido '.' é assumido.
  Diretórios pais são criados conforme necessário no repositório.
  Se CAMINHO é um diretório, seu conteúdo será adicionado diretamente
  abaixo de URL.

Opções válidas:
  -q [--quiet]             : imprime o mínimo possível
  -N [--non-recursive]     : opera somente em um diretório
…
```

Colocando dados em seu Repositório

Há dois modos de colocar novos arquivos em seu repositório Subversion: **svn import** e **svn add**. Discutiremos **svn import** aqui e **svn add** mais adiante neste capítulo quando analisarmos um dia típico com o Subversion.

svn import

O comando **svn import** é um modo rápido para copiar uma árvore de arquivos não versionada em um repositório, criando diretórios intermediários quando necessário. **svn import** não requer uma cópia de trabalho, e seus arquivos são imediatamente submetidos ao repositório. Isto é tipicamente usado quando você tem uma árvore de arquivos existente que você quer monitorar em seu repositório Subversion. Por exemplo:

```
$ svnadmin create /usr/local/svn/newrepos
```

```
$ svn import mytree file:///usr/local/svn/newrepos/some/project \
             -m "Importação inicial"
Adicionando    mytree/foo.c
Adicionando    mytree/bar.c
Adicionando    mytree/subdir
Adicionando    mytree/subdir/quux.h

Commit da revisão 1.
```

O exemplo anterior copiou o conteúdo do diretório `mytree` no diretório `some/project` no repositório:

```
$ svn list file:///usr/local/svn/newrepos/some/project
bar.c
foo.c
subdir/
```

Note que após a importação terminar, a árvore inicial *não* está convertida em uma cópia de trabalho. Para começar a trabalhar, você ainda precisa obter(**svn checkout**) uma nova cópia de trabalho da árvore.

Layout de repositório recomendado

Enquanto a flexibilidade do Subversion permite que você organize seu repositório da forma que você escolher, nós recomendamos que você crie um diretório `trunk` para armazenar a “linha principal” de desenvolvimento, um diretório `branches` para conter cópias ramificadas, e um diretório `tags` para conter cópias rotuladas, por exemplo:

```
$ svn list file:///usr/local/svn/repos
/trunk
/branches
/tags
```

Você aprenderá mais sobre tags e branches no Capítulo 4, *Fundir e Ramificar*. Para detalhes e como configurar múltiplos projetos, veja “Repository Layout” e “Planejando a Organização do Repositório” para ler mais sobre “raízes dos projetos”.

Checkout Inicial

Na maioria das vezes, você começa a usar um repositório Subversion fazendo um *checkout* de seu projeto. Fazer um checkout de um repositório cria uma “cópia de trabalho” em sua máquina local. Esta cópia contém o `HEAD` (revisão mais recente) do repositório Subversion que você especificou na linha de comando:

```
$ svn checkout http://svn.collab.net/repos/svn/trunk
A    trunk/Makefile.in
A    trunk/ac-helpers
A    trunk/ac-helpers/install.sh
A    trunk/ac-helpers/install-sh
A    trunk/build.conf
…
Gerado cópia de trabalho para revisão 8810.
```

O que há em um Nome?

O Subversion tenta arduamente não limitar o tipo de dado que você pode colocar sob controle de versão. O conteúdo dos arquivos e valores de propriedades são armazenados e transmitidos como dados binários, e a seção “Tipo de Conteúdo do Arquivo” diz-lhe como dar ao Subversion uma dica de que operações “textuais” não têm sentido para um arquivo em particular. Há umas poucas ocasiões, porém, onde o Subversion coloca restrições sobre as informações nele armazenadas.

O Subversion manipula internamente determinados fragmentos de dados—por exemplo, nomes de propriedades, nomes de caminhos, e mensagens de log—como Unicode codificado em UTF-8. Porém, isto não quer dizer que todas suas interações com o Subversion devam envolver UTF-8. Como uma regra geral, os clientes Subversion graciosa e transparentemente manipularão as conversões entre UTF-8 e o sistema de codificação em uso em seu computador, caso tal conversão possa ser feita de forma que faça sentido (o que é o caso das codificações mais comuns em uso hoje).

Adicionalmente, caminhos de arquivos são usados como valores de atributos XML nas trocas WebDAV, bem como em alguns arquivos internamente mantidos pelo Subversion. Isto significa que os caminhos de arquivos podem somente conter caracteres aceitos no XML (1.0). Subversion também proíbe os caracteres TAB, CR, e LF em nomes de caminhos para prevenir que caminhos sejam quebrados nos diffs, ou em saídas de comandos como svn log ou svn status.

Embora pareça que há muito o que recordar, na prática estas limitações raramente são um problema. Enquanto suas configurações regionais são compatíveis com UTF-8, e você não usar caracteres de controle nos nomes dos caminhos, você não terá problemas na comunicação com o Subversion. O cliente de linha de comando dá um pouco de ajuda extra—ele automaticamente adiciona informações de escape para os caracteres ilegais nos caminhos em URLs que você digita, para criar versões “legalmente corretas” para uso interno quando necessário.

Embora os exemplos acima efetuem o checkout do diretório trunk, você pode facilmente efetuar o checkout em qualquer nível de subdiretórios de um repositório especificando o subdiretório na URL do checkout:

```
$ svn checkout \
      http://svn.collab.net/repos/svn/trunk/subversion/tests/cmdline/
A    cmdline/revert_tests.py
A    cmdline/diff_tests.py
A    cmdline/autoprop_tests.py
A    cmdline/xmltests
A    cmdline/xmltests/svn-test.sh
…
Gerado cópia de trabalho para revisão 8810.
```

Uma vez que o Subversion usa um modelo “copiar-modificar-fundir” ao invés de “travar-modificar-destravar” (veja “Modelos de Versionamento”), você pode iniciar por fazer alterações nos arquivos e diretórios em sua cópia de trabalho. Sua cópia de trabalho é igual a qualquer outra coleção de arquivos e diretórios em seu sistema. Você pode editá-los, alterá-los e movê-los, você pode até mesmos apagar toda sua cópia de trabalho e esquecê-la.

Apesar de sua cópia de trabalho ser “igual a qualquer outra coleção de arquivos e diretórios em seu sistema”, você pode editar os arquivos a vontade, mas tem que informar o Subversion sobre *tudo o mais* que você fizer. Por exemplo, se você quiser copiar ou mover um item em uma cópia de trabalho, você deve usar os comandos **svn copy** or **svn move** em vez dos comandos copiar e mover fornecidos por ser sistema operacional. Nós falaremos mais sobre eles posteriormente neste capítulo.

A menos que você esteja pronto para submeter a adição de novos arquivos ou diretórios, ou modificações nos já existentes, não há necessidade de continuar a notificar o servidor Subversion que você tenha feito algo.

O que há no diretório `.svn`?

Cada diretório em uma cópia de trabalho contém uma área administrativa, um subdiretório nomeado `.svn`. Normalmente, comandos de listagem de diretórios não mostrarão este subdiretório, mas este é um diretório importante. Faça o que fizer, não apague ou modifique nada nesta área administrativa! O Subversion depende dela para gerenciar sua cópia de trabalho.

Se você remover o subdiretório `.svn` acidentalmente, o modo mais fácil de resolver o problema é remover todo o conteúdo do diretório (uma exclusão normal pelo sistema, não **svn delete**), então executar **svn update** a partir do diretório pai. O cliente Subversion fará novamente o download do diretório que você excluiu, bem como uma nova área `.svn`.

Além de você certamente poder obter uma cópia de trabalho com a URL do repositório como único argumento, você também pode especificar um diretório após a URL do repositório. Isto coloca sua cópia de trabalho no novo diretório que você informou. Por exemplo:

```
$  svn checkout http://svn.collab.net/repos/svn/trunk subv
A    subv/Makefile.in
A    subv/ac-helpers
A    subv/ac-helpers/install.sh
A    subv/ac-helpers/install-sh
A    subv/build.conf
…
Gerado cópia de trabalho para revisão 8810.
```

Isto colocará sua cópia de trabalho em um diretório chamado `subv` em vez de um diretório chamado `trunk` como fizemos anteriormente. O diretório `subv` será criado se ele não existir.

Desabilitando o Cache de Senhas

Quando você realiza uma operação no Subversion que requer autenticação, por padrão o Subversion mantém suas credenciais de autenticação num cache em disco. Isto é feito por conveniência, para que você não precise continuamente ficar redigitando sua senha em operações futuras. Se você estiver preocupado com o fato de o Subversion manter um cache de suas senhas, [1] você pode desabilitar o cache de forma permanente ou analisando caso a caso.

Para desabilitar o cache de senhas para um comando específico uma vez, passe a opção `--no-auth-cache` na linha de comando. Para desabilitar permanentemente o cache, você pode adicionar a linha `store-passwords = no` no arquivo de configuração local do seu Subversion. Veja “Armazenando Credenciais no Cliente” para maiores detalhes.

Autenticando como um Usuário Diferente

Uma vez que por padrão o Subversion mantém um cache com as credenciais de autenticação (tanto usuário quanto senha), ele convenientemente se lembra que era você estava ali na última vez que você modificou sua cópia de trabalho. Mas algumas vezes isto não é útil — particularmente se você estava trabalhando numa cópia de trabalho compartilhada, como um diretório de configuração do sistema ou

[1] É claro, você não está terrivelmente preocupado — primeiro porque você sabe que você não pode *realmente* deletar nada do Subversion e, em segundo lugar, porque sua senha do Subversion não é a mesma que as outras três milhões de senhas que você tem, certo? Certo?

o documento raiz de um servidor web. Neste caso, apenas passe a opção `--username` na linha de comando e o Subversion tentará autenticar como aquele usuário, pedindo uma senha se necessário.

Ciclo Básico de Trabalho

O Subversion tem diversos recursos, opções, avisos e sinalizações, mas no básico do dia-a-dia, é mais provável que você utilize apenas uns poucos destes recursos. Nesta seção vamos abordar as coisas mais comuns que você de fato pode fazer com o Subversion no decorrer de um dia de trabalho comum.

Um ciclo básico de trabalho é parecido com:

- Atualizar sua cópia de trabalho
  - **svn update**
- Fazer alterações
  - **svn add**
  - **svn delete**
  - **svn copy**
  - **svn move**
- Verificar suas alterações
  - **svn status**
  - **svn diff**
- Possivelmente desfazer algumas alterações
  - **svn revert**
- Resolver conflitos (combinar alterações de outros)
  - **svn update**
  - **svn resolved**
- Submeter suas alterações
  - **svn commit**

Atualizando Sua Cópia de Trabalho

Ao trabalhar num projeto em equipe, você vai querer atualizar sua cópia de trabalho para receber quaisquer alterações feitas por outros desenvolvedores do projeto desde sua última atualização. Use **svn update** para deixar sua cópia de trabalho em sincronia com a última revisão no repositório.

```
$ svn update
U  foo.c
U  bar.c
Updated to revision 2.
```

Neste caso, alguém submeteu modificações em `foo.c` e `bar.c` desde a última vez que você atualizou, e o Subversion atualizou sua cópia de trabalho para incluir estas modificações.

Quando o servidor envia as alterações para sua cópia de trabalho por meio do **svn update**, uma letra é exibida como código próximo de cada item para que você saiba que ações o Subversion executou para deixar sua cópia de trabalho atualizada. Para conferir o que essas letras significam, veja svn update.

Fazendo Alterações em Sua Cópia de Trabalho

Agora você já pode trabalhar e fazer alterações em sua cópia de trabalho. É comumente mais conveniente optar por fazer uma alteração (ou conjunto de alterações) discreta, como escrever um novo recurso, corrigir um bug, etc. Os comandos do Subversion que você usará aqui são **svn add**, **svn delete**, **svn copy**, **svn move**, e **svn mkdir**. No entanto, se você está meramente editando arquivos que já se encontram no Subversion, você pode não precisar usar nenhum destes comandos para registrar suas alterações.

Há dois tipos de alterações que você pode fazer em sua cópia de trabalho: alterações nos arquivos e alterações na árvore. Você não precisa avisar ao Subversion que você pretende modificar um arquivo; apenas faça suas alterações usando seu editor de texto, suite de escritório, programa gráfico, ou qualquer outra ferramenta que você use normalmente. O Subversion automaticamente irá identificar que arquivos foram modificados, ele também vai manipular arquivos binários da mesma forma que manipula arquivos de texto—e tão eficientemente quanto. Para alterações na árvore, você pode solicitar ao Subversion que "marque" os arquivos e diretórios para remoção, adição, cópia ou movimentação agendada. Estas alterações terão efeito imediatamente em sua cópia de trabalho, mas nenhuma adição ou remoção vai acontecer no repositório até que você registre tais alterações.

Aqui está uma visão geral dos cinco subcomandos do Subversion que você vai usar mais frequentemente para fazer alterações na árvore.

Versionando links simbólicos

Em plataformas não-Windows, o Subversion é capaz de versionar arquivos do tipo especial *link simbólico* (ou, "symlink"). Um link simbólico é um arquivo que funciona como uma espécie de referência transparente para alguns outros objetos no sistema de arquivos, permitindo que programas leiam e escrevam nestes objetos indiretamente através da execução destas operações no link simbólico em si.

Quando um link simbólico é registrado em um repositório Subversion, o Subversion se lembra que o arquivo é de fato um symlink, bem como também se lembra do objeto ao qual o link "aponta". Quando o link simbólico sofre um checkout em outra cópia de trabalho em um sistema não-Windows, o Subversion recria um link simbólico no nível do sistema de arquivos real a partir do symlink versionado. Mas isto de forma nenhuma limita a usabilidade das cópias de trabalho em sistemas como o Windows que não suportam links simbólicos. Nesses sistemas, o Subversion simplesmente cria um arquivo de texto normal que cujo conteúdo é o caminho para o qual o link original aponta. Apesar deste arquivo não poder ser usado como link simbólico num sistema Windows, ele também não proíbe os usuários Windows de executarem suas outras atividades relacionadas ao Subversion.

svn add foo

Agenda o arquivo, diretório, ou link simbólico `foo` para ser adicionado ao repositório. Na próxima vez que você der um commit, `foo` passará a fazer parte do diretório pai onde estiver. Veja que se `foo` for um diretório, tudo que estiver dentro de `foo` será marcado para adição. Se você quiser adicionar apenas o diretório `foo` em si, inclua a opção `--non-recursive (-N)`.

svn delete foo

Agenda o arquivo, diretório, ou link simbólico `foo` para ser excluído do repositório. Se `foo` for um arquivo ou link, ele é imediatamente removido de sua cópia de trabalho. Se `foo` for um diretório, ele não é excluído, mas o Subversion o deixa agendado para exclusão. Quando você der commit em suas alterações, `foo` será inteiramente removido de sua cópia de trabalho e do repositório. [2]

[2]Claro que nada é completamente excluído do repositório—mas apenas da versão HEAD do repositório. Você pode trazer de volta qualquer coisa que você tenha excluído dando um checkout (ou atualizando sua cópia de trabalho) para uma revisão anterior àquela em que você tenha feito a exclusão. Veja também "Ressucitando Itens Excluídos".

svn copy foo bar

Cria um novo item `bar` como uma duplicata de `foo` e agenda `bar` automaticamente para adição. Quando `bar` for adicionado ao repositório no próximo commit, o histórico da cópia é gravado (como vindo originalmente de `foo`). **svn copy** não cria diretórios intermediários.

svn move foo bar

Este comando é exatamente o mesmo que **svn copy foo bar; svn delete foo**. Isto é, `bar` é agendado para ser adicionado como uma cópia de `foo`, e `foo` é agendado para remoção. **svn move** não cria diretórios intermediários.

svn mkdir blort

Este comando é exatamente o mesmo que se executar **mkdir blort; svn add blort**. Isto é, um novo diretório chamado `blort` é criado e agendado para adição.

Modificando o Repositório Sem uma Cópia de Trabalho

Há algumas formas de registrar alterações imediatamente na árvore do repositório. Isto apenas acontece quando um subcomando está operando diretamente numa URL, ao invés de na cópia de trabalho. Em particular, usos específicos de **svn mkdir**, **svn copy**, **svn move** e **svn delete** trabalham com URLs (e não esqueça que o **svn import** sempre faz alterações em uma URL).

Operações em URL funcionam desta maneira porque os comandos que operam em uma cópia de trabalho podem usar a cópia de trabalho como uma espécie de "área de teste" onde executar suas alterações antes de registrá-las efetivamente no repositório. Os comandos que operam em URLs não dispõem deste luxo, então quando você opera diretamente em uma URL, quaisquer das ações acima resultam em commits imediatos.

Verificando Suas Alterações

Tendo terminado de fazer suas alterações, você precisa registrá-las no repositório, mas antes de fazer isso, é quase sempre uma boa idéia conferir exatamente que alterações você fez. Ao verificar suas alterações antes de dar commit, você pode criar uma mensagem de log bem mais adequada. Você também pode descobrir se não modificou um arquivo inadvertidamente, e então ter a oportunidade de reverter essas modificações antes de dar commit. Você pode ter uma visão geral das alterações que você fez usando **svn status**, e obter os detalhes sobre essas alterações usando **svn diff**.

Olha Mãe! Sem a Rede!

Os comandos **svn status**, **svn diff**, e **svn revert** podem ser usados sem nenhum acesso a rede mesmo se seu repositório *for* disponibilizado em rede. Isto facilitar a gerência de suas alterações em curso quando você estiver sem conexão de rede, como enquanto estiver voando de avião, andando de trem ou mesmo usando seu notebook na praia.[3]

O Subversion faz isso mantendo caches privados das versões intactas de cada arquivo sob controle de versão dentro dos diretórios das áreas administrativas `.svn`. Isto permite ao Subversion reportar—e reverter—modificações locais nestes arquivos *sem precisar acessar a rede*. Este cache (chamado de "texto-base") também possibilita ao Subversion enviar as modificações locais do usuário durante um commit ao servidor como um arquivo compactado *delta* (ou "diferença") sobre a versão original do arquivo. Ter este cache representa um grande benefício—mesmo se você tiver uma conexão rápida de rede, é muito mais rápido enviar apenas as modificações do arquivo que enviar todo o arquivo para o servidor.

O Subversion está sendo otimizado para ajudar você com esta tarefa e é capaz de fazer muitas coisas sem se comunicar com o repositório. Em particular, sua cópia de trabalho contém um cache

[3]Daquelas que não tenham acesso sem-fio. Achou que ia nos pegar, hein?

escondido com uma cópia “intacta” de cada arquivo sob controle de versão dentro da área `.svn`. Por isso, o Subversion pode rapidamente lhe mostrar como seus arquivos de trabalho mudaram, ou mesmo permitir a você desfazer suas alterações sem contactar o repositório.

Obtendo uma visão geral de suas alterações

Para ter uma visão geral de suas modificações, você vai usar o comando **svn status**. Você provavelmente vai usar mais o comando **svn status** do que qualquer outro comando do Subversion.

> **Usuários CVS: Segurem o Update!**
>
> Você provavelmente costuma usar **cvs update** para ver que alterações você fez em sua cópia de trabalho. O **svn status** lhe dará toda a informação de que você precisa sobre o que mudou em sua cópia de trabalho—sem acessar o repositório ou potencialmente incorporar novas alterações publicadas por outros usuários.
>
> No Subversion, **update** faz apenas isso—ele atualiza sua cópia de trabalho com quaisquer alterações registradas no repositório desde a última vez que você atualizou sua cópia de trabalho. Você deve quebrar o hábito de usar o comando **update** para ver que alterações locais você fez.

Se você executar **svn status** no topo de cópia de trabalho sem argumentos, ele irá detectar todas as alterações de arquivos e árvores que você fez. Abaixo estão uns poucos exemplos dos códigos mais comuns de estado que o **svn status** pode retornar. (Note que o texto após # não é exibido pelo **svn status**.)

```
A       stuff/loot/bloo.h   # arquivo agendado para adição
C       stuff/loot/lump.c   # arquivo em conflito a partir de um update
D       stuff/fish.c        # arquivo agendado para exclusão
M       bar.c               # conteúdo em bar.c tem alterações locais
```

Neste formato de saída **svn status** exibe seis colunas de caracteres, seguidas de diversos espaços em branco, seguidos por um nome de arquivo ou diretório. A primeira coluna indica o estado do arquivo ou diretório e/ou seu conteúdo. Os códigos listados são:

`A item`
: O arquivo, diretório, ou link simbólico `item` está agendado para ser adicionado ao repositório.

`C item`
: O arquivo `item` está em um estado de conflito. Isto é, as modificações recebidas do servidor durante um update se sobrepõem às alterações locais feitas por você em sua cópia de trabalho. Você deve resolver este conflito antes de submeter suas alterações ao repositório.

`D item`
: O arquivo, diretório, ou link simbólico `item` está agendado para ser excluído do repositório.

`M item`
: O conteúdo do arquivo `item` foi modificado.

Se você passar um caminho específico para o **svn status**, você vai obter informação apenas sobre aquele item:

```
$ svn status stuff/fish.c
D      stuff/fish.c
```

O **svn status** também tem uma opção `--verbose (-v)`, a qual vai lhe mostrar o estado de *cada* item em sua cópia de trabalho, mesmo se não tiver sido modificado:

```
$ svn status -v
M               44        23    sally     README
                44        30    sally     INSTALL
M               44        20    harry     bar.c
                44        18    ira       stuff
                44        35    harry     stuff/trout.c
D               44        19    ira       stuff/fish.c
                44        21    sally     stuff/things
A                0         ?     ?        stuff/things/bloo.h
                44        36    harry     stuff/things/gloo.c
```

Esta é a “forma estendida” da saída do **svn status**. As letras na primeira coluna significam o mesmo que antes, mas a segunda coluna mostra a revisão de trabalho do item. A terceira e quarta coluna mostram a revisão na qual o item sofreu a última alteração, e quem o modificou.

Nenhuma das execuções anteriores de **svn status** contactam o repositório—ao invés disso, elas comparam os metadados no diretório `.svn` com a cópia de trabalho. Finalmente, existe a opção `--show-updates (-u)`, que se conecta ao repositório e adiciona informação sobre as coisas que estão desatualizadas:

```
$ svn status -u -v
M      *        44        23    sally     README
M               44        20    harry     bar.c
       *        44        35    harry     stuff/trout.c
D               44        19    ira       stuff/fish.c
A                0         ?     ?        stuff/things/bloo.h
Status against revision:   46
```

Perceba os dois asteriscos: se você executar **svn update** neste ponto, você deverá receber alterações nos arquivos `README` e `trout.c`. Isto lhe dá alguma informação bastante útil—você vai precisar atualizar e obter as modificações do servidor no arquivo `README` antes de executar um commit, ou o repositório vai rejeitar sua submissão por ter estar desatualizada. (Mais sobre este assunto mais tarde.)

O **svn status** pode exibir muito mais informação sobre os arquivos e diretórios em sua cópia de trabalho do que o que mostramos aqui—para uma descrição exaustiva do svn status e de sua saída, veja svn status.

Examinando os detalhes de suas alterações locais

Outra forma de examinar suas alterações é com o comando **svn diff**. Você pode verificar *exatamente* como você modificou as coisas executando **svn diff** sem argumentos, o que exibe as modificações de arquivo no *formato diff unificado*:

```
$ svn diff
Index: bar.c
===================================================================
--- bar.c (revision 3)
+++ bar.c (working copy)
@@ -1,7 +1,12 @@
+#include <sys/types.h>
+#include <sys/stat.h>
+#include <unistd.h>
+
+#include <stdio.h>

 int main(void) {
```

```
-  printf("Sixty-four slices of American Cheese...\n");
+  printf("Sixty-five slices of American Cheese...\n");
 return 0;
 }

Index: README
===================================================================
--- README (revision 3)
+++ README (working copy)
@@ -193,3 +193,4 @@
+Note to self:  pick up laundry.

Index: stuff/fish.c
===================================================================
--- stuff/fish.c (revision 1)
+++ stuff/fish.c (working copy)
-Welcome to the file known as 'fish'.
-Information on fish will be here soon.

Index: stuff/things/bloo.h
===================================================================
--- stuff/things/bloo.h (revision 8)
+++ stuff/things/bloo.h (working copy)
+Here is a new file to describe
+things about bloo.
```

O comando **svn diff** produz esta saída comparando seus arquivos de trabalho com a cópia "intacta" em cache que fica dentro da área `.svn`. Os arquivos marcados para adição são exibidos com todo o texto adicionado, e os arquivos marcados para exclusão são exibidos com todo o texto excluído.

A saída é exibida no formato diff unificado. Isto é, linhas removidas são iniciadas com `-` e linhas adicionadas são iniciadas com `+`. O **svn diff** também exibe o nome do arquivo e uma informação de deslocamento (*offset*) que é útil para o programa **patch**, de forma que você pode gerar "atualizações" (*patches*) redirecionando a saída do diff para um arquivo:

```
$ svn diff > patchfile
```

Você pode, por exemplo, enviar um arquivo de atualização por e-mail para outro desenvolvedor para revisão ou teste prévio antes de submeter.

O Subversion usa seu mecanismo interno de diff, o qual gera o formato diff unificado, por padrão. Se você quiser esta saída em um formato diferente, especifique um programa diff externo usando `--diff-cmd` e passe as opções de sua preferência para ele com `--extensions (-x)`. Por exemplo, para ver as diferenças locais no arquivo `foo.c` no formato de saída de contexto ao mesmo tempo ignorando diferenças de maiúsculas e minúsculas, você poderia executar **svn diff --diff-cmd /usr/bin/diff --extensions '-i' foo.c**.

Desfazendo Modificações de Trabalho

Suponha que ao ver a saída do **svn diff** você decida que todas as modificações que você fez em um certo arquivo foram equivocadas. Talvez você não tivesse que ter modificado o arquivo como um todo, ou talvez você veja que acabará sendo mais fácil fazer outras modificações começando tudo de novo.

Esta é uma oportunidade perfeita para usar o comando **svn revert**:

```
$ svn revert README
Reverted 'README'
```

O Subversion reverte o arquivo para seu estado antes da alteração sobrescrevendo-o com a cópia "intacta" que está na área `.svn`. Mas atente também que **svn revert** pode desfazer *quaisquer* operações agendadas—por exemplo, você pode decidir que você não quer mais adicionar um novo arquivo:

```
$ svn status foo
?      foo

$ svn add foo
A         foo

$ svn revert foo
Reverted 'foo'

$ svn status foo
?      foo
```

svn revert *`ITEM`* tem exatamente o mesmo efeito de se excluir o *`ITEM`* de sua cópia de trabalho e então executar **svn update -r BASE** *`ITEM`*. Porém, se você estiver revertendo um arquivo, o **svn revert** tem uma notável diferença—ele não precisa se comunicar com o repositório para restaurar o seu arquivo.

Ou talvez você tenha removido um arquivo do controle de versão por engano:

```
$ svn status README
       README

$ svn delete README
D         README

$ svn revert README
Reverted 'README'

$ svn status README
       README
```

Resolvendo Conflitos (Combinando Alterações de Outros)

Já vimos como o **svn status -u** pode prever conflitos. Suponha que você execute **svn update** e aconteça algo interessante:

```
$ svn update
U  INSTALL
G  README
C  bar.c
Updated to revision 46.
```

Os códigos `U` e `G` não são motivo de preocupação; esses arquivos absorveram as alterações do repositório normalmente. O arquivo marcado com `U` não continha modificações locais mas foi atualizado (*Updated*) com as modificações do repositório. O `G` vem de *merGed*, o que significa que o arquivo local continha alterações inicialmente, e que também houve alterações vindas do repositório, que no entanto não se sobrepuseram às alterações locais.

Já o `C` significa Conflito. Isto quer dizer que as modificações do servidor se sobrepõem com as suas próprias, e que agora você precisa escolher entre elas manualmente.

Sempre que um conflito ocorre, normalmente acontecem três coisas para ajudar você a compreender e resolver este conflito:

- O Subversion exibe um `C` durante o update, e lembra que o arquivo está num estado de conflito.
- Se o Subversion considerar que o arquivo é mesclável, ele põe *marcações de conflito*—strings de texto especiais que delimitam os “lados” do conflito—dentro do arquivo para mostrar visivelmente as áreas de sobreposição. (O Subversion usa a propriedade `svn:mime-type` para decidir se um arquivo é passível de combinação contextual de linhas. Veja “Tipo de Conteúdo do Arquivo” para saber mais.)
- Para cada arquivo em conflito, o Subversion mantém três arquivos extras não-versionados em sua cópia de trabalho:

  `filename.mine`
  : Este é o seu arquivo como o que existia em sua cópia de trabalho antes de você atualizá-la—isto é, sem as marcações de conflito. Este arquivo tem apenas as suas últimas alterações feitas nele. (Se o Subversion considerar que o arquivo não é mesclável, então o arquivo `.mine` não é criado, uma vez que seria idêntico ao arquivo de trabalho.)

  `filename.rOLDREV`
  : Este é o arquivo que estava na revisão `BASE` antes de você atualizar sua cópia de trabalho. Isto é, o arquivo em que você pegou do repositório antes de fazer suas últimas alterações.

  `filename.rNEWREV`
  : Este é o arquivo que seu cliente Subversion acabou de receber do servidor quando você atualizou sua cópia de trabalho. Este arquivo corresponde à revisão `HEAD` do repositório.

  Aqui, `OLDREV` é o número de revisão do arquivo em seu diretório `.svn` e `NEWREV` é o número de revisão do repositório `HEAD`.

Como exemplo, Sally faz modificações no arquivo `sandwich.txt` no repositório. Harry acabou de alterar o arquivo em sua cópia de trabalho e o submeteu. Sally atualiza sua cópia de trabalho antes de executar um commit o que resulta em um conflito:

```
$ svn update
C  sandwich.txt
Updated to revision 2.
$ ls -1
sandwich.txt
sandwich.txt.mine
sandwich.txt.r1
sandwich.txt.r2
```

Neste ponto, o Subversion *não* vai deixar que você submeta o arquivo `sandwich.txt` até que os três arquivos temporários sejam removidos.

```
$ svn commit -m "Add a few more things"
svn: Commit failed (details follow):
svn: Aborting commit: '/home/sally/svn-work/sandwich.txt' remains in conflict
```

Se você tiver um conflito, você precisa fazer uma dessas três coisas:

- Mesclar o texto conflituoso “na mão” (examinando e editando as marcações de conflito dentro do arquivo).
- Fazer uma cópia de um dos arquivos temporários em cima de seu arquivo de trabalho.
- Executar **svn revert <filename>** para se desfazer de todas as suas modificações locais.

Uma vez que você tenha resolvido o conflito, você precisa informar isto ao Subversion executando **svn resolved**. Isso remove os três arquivos temporários e o Subversion não mais considera o arquivo como estando em conflito.[4]

```
$ svn resolved sandwich.txt
Resolved conflicted state of 'sandwich.txt'
```

Mesclando Conflitos na Mão

Mesclar conflitos na mão pode ser algo bem intimidante na na primeira vez que você tentar, mas com um pouco de prática, pode ser tornar tão fácil quanto andar de bicicleta.

Veja um exemplo. Por um problema de comunicação, você e Sally, sua colaboradora, ambos editam o arquivo `sandwich.txt` ao mesmo tempo. Sally submete suas alterações, e quando você atualizar sua cópia de trabalho, você terá um conflito e precisará editar o arquivo `sandwich.txt` para resolvê-los. Primeiro, vamos dar uma olhada no arquivo:

```
$ cat sandwich.txt
Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
<<<<<<< .mine
Salami
Mortadella
Prosciutto
=======
Sauerkraut
Grilled Chicken
>>>>>>> .r2
Creole Mustard
Bottom piece of bread
```

As strings de sinais de menor, sinais de igual, e sinais de maior são marcações de conflito, e não fazem parte atualmente dos dados em conflito. Você geralmente quer garantir que estes sinais sejam removidos do arquivo antes de seu próximo commit. O texto entre os dois primeiros conjuntos de marcações é composto pelas alterações que você fez na área do conflito:

```
<<<<<<< .mine
Salami
Mortadella
Prosciutto
=======
```

O texto entre o segundo e terceiro conjuntos de marcações de conflito é o texto das alterações submetidas por Sally:

```
=======
Sauerkraut
Grilled Chicken
>>>>>>> .r2
```

[4]Você sempre pode remover os arquivos temporários você mesmo, mas você vai realmente querer fazer isso quando o Subversion pode fazer por você? Nós achamos que não.

Normalmente você não vai querer apenas remover as marcações e as alterações de Sally—ela ficaria terrivelmente surpresa quando o sanduíche chegar e não for o que ela queria. Então este é o momento em que você pega o telefone ou atravessa o escritório e explica para Sally o que você não gosta de sauerkraut como iguaria italiana.[5] Uma vez que vocês tenham chegado a um acordo sobre as alterações que serão mantidas, edite seu arquivo e remova as marcações de conflito.

```
Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
Salami
Mortadella
Prosciutto
Creole Mustard
Bottom piece of bread
```

Agora execute **svn resolved**, e você estará pronto para submeter suas alterações:

```
$ svn resolved sandwich.txt
$ svn commit -m "Go ahead and use my sandwich, discarding Sally's edits."
```

Veja que o **svn resolved**, ao contrário muitos dos outros comandos que abordamos neste capítulo, precisa de um argumento. Em todo caso, você vai querer ter cuidado e só executar **svn resolved** quando tiver certeza de ter resolvido o conflito em seu arquivo—quando os arquivos temporários forem removidos, o Subversion vai deixar que você submeta o arquivo ainda que ele permaneça com marcações de conflito.

Se você se confundiu ao editar o arquivo conflituoso, você sempre pode consultar os três arquivos que o Subversion cria para você em sua cópia de trabalho—inclusive o seu arquivo como era antes de você atualizar. Você ainda pode usar uma ferramenta interativa de mesclagem de terceiros para examinar esses três arquivos.

Copiando um Arquivo em Cima de Seu Arquivo de Trabalho

Se você tiver um conflito e decidir que quer descartar suas modificações, você pode meramente copiar um dos arquivos temporários criados pelo Subversion em cima do arquivo de sua cópia de trabalho:

```
$ svn update
C  sandwich.txt
Updated to revision 2.
$ ls sandwich.*
sandwich.txt  sandwich.txt.mine  sandwich.txt.r2  sandwich.txt.r1
$ cp sandwich.txt.r2 sandwich.txt
$ svn resolved sandwich.txt
```

Punting: Usando o svn revert

Se você tiver um conflito, e após examinar, decidir que prefere descartar suas alterações e começar a editar outras coisas, apenas reverta suas alterações:

```
$ svn revert sandwich.txt
Reverted 'sandwich.txt'
```

[5] E se você lhes disser isso, eles podem muito bem expulsar você para fora da cidade num minuto.

```
$ ls sandwich.*
sandwich.txt
```

Perceba que ao reverter um arquivo em conflito, você não precisa executar **svn resolved**.

Registrando Suas Alterações

Finalmente! Suas edições estão concluídas, você mesclou todas as alterações do servidor, e agora está pronto para registrar suas alterações no repositório.

O comando **svn commit** envia todas as suas modificações para o servidor. Quando você registra uma alteração, você precisa informar uma *mensagem de log*, descrevendo sua alteração. Sua mensagem de log será anexada à nova revisão que você criar. Se sua mensagem de log for breve, você pode querer escrevê-la na própria linha de comando usando a opção `--message` (ou `-m`):

```
$ svn commit -m "Corrected number of cheese slices."
Sending        sandwich.txt
Transmitting file data .
Committed revision 3.
```

No entanto, se você estiver escrevendo sua mensagem conforme for trabalhando, você pode querer informar ao Subversion para obter a mensagem a partir de um arquivo indicando-o com a opção `--file (-F)`:

```
$ svn commit -F logmsg
Sending        sandwich.txt
Transmitting file data .
Committed revision 4.
```

Se você não especificar a opção `--message` nem a `--file`, o Subversion vai abrir automaticamente seu editor de texto preferido (veja a seção `editor-cmd` em “Configuração”) para compor a mensagem de log.

Se você estiver escrevendo a mensagem de log em seu editor neste ponto e decidir cancelar seu commit, você pode apenas sair do seu editor sem salvar suas alterações. Se você já salvou sua mensagem de log, simplesmente exclua o texto, salve novamente, então feche o programa.

```
$ svn commit
Waiting for Emacs...Done

Log message unchanged or not specified
a)bort, c)ontinue, e)dit
a
$
```

O repositório não sabe nem mesmo se importa se suas modificações sequer fazem sentido como um todo; ele apenas garante que ninguém mais tenha modificado nada dos mesmos arquivos que você enquanto você não estava olhando. Se alguém *tiver* feito isso, o commit todo irá falhar com uma mensagem informando a você que um ou mais de seus arquivos está desatualizado:

```
$ svn commit -m "Add another rule"
Sending        rules.txt
svn: Commit failed (details follow):
svn: Your file or directory 'sandwich.txt' is probably out-of-date
```

…

(Os dizeres exatos desta mensagem de erro dependem de qual protocolo de rede e qual servidor você está usando, mas a idéia é a mesma em todos os casos.)

Neste ponto, você precisa executar **svn update**, lidar com quaisquer mesclagens ou conflitos resultantes, e tentar executar o commit novamente.

Isto conclui o ciclo básico de trabalho no uso do Subversion. Há muitos outros recursos no Subversion que você pode usar para gerenciar seu repositório e sua cópia de trabalho, mas muito de seu uso cotidiano do Subversion vai envolver apenas os comandos que discutimos neste capítulo. Vamos, entretanto, abordar mais uns poucos comandos que você também pode usar bastante.

Examinando o Histórico

O seu repositório Subversion é como uma máquina do tempo. Ele mantém um registro de cada modificação submetida, e permite a você explorar este histórico examinando versões anteriores de seus arquivos e diretórios bem como os metadados a eles relacionados. Com apenas um comando do Subversion, você pode deixar o repositório (ou restaurar uma cópia de trabalho existente) exatamente como ele era em uma certa data ou em um número de revisão no passado. Porém, algumas vezes você só quer *revisitar* o passado ao invés de *retornar* ao passado.

Há uma série de comandos que podem lhe mostrar dados históricos do repositório:

svn log
: Exibe bastante informação: mensagens de log com informações de data e autor anexadas às revisões, e quais caminhos mudaram em cada revisão.

svn diff
: Exibe detalhes a nível das linhas de uma alteração em particular.

svn cat
: Recupera um arquivo como ele era em uma dada revisão e exibe seu conteúdo na tela.

svn list
: Exibe os arquivos em um diretório para uma dada revisão.

Gerando uma lista de alterações históricas

Para encontrar informação acerca do histórico de um arquivo ou diretório, use o comando **svn log**. O **svn log** vai lhe dar um registro de quem fez alterações em um arquivo ou diretório, em qual revisão houve a mudança, a data e hora daquela revisão, e, se for informada, a mensagem de log associada a esse registro.

```
$ svn log
------------------------------------------------------------------------
r3 | sally | Mon, 15 Jul 2002 18:03:46 -0500 | 1 line

Added include lines and corrected # of cheese slices.
------------------------------------------------------------------------
r2 | harry | Mon, 15 Jul 2002 17:47:57 -0500 | 1 line

Added main() methods.
------------------------------------------------------------------------
r1 | sally | Mon, 15 Jul 2002 17:40:08 -0500 | 1 line

Initial import
------------------------------------------------------------------------
```

Veja que as mensagens de log são exibidas em *ordem cronológica inversa* por padrão. Se você quiser ver um intervalo de revisões em uma ordem específica, ou apenas uma única revisão, utilize a opção `--revision (-r)`:

```
$ svn log -r 5:19    # exibe os logs de 5 a 19 em ordem cronológica

$ svn log -r 19:5    # exibe os logs de 5 a 19 na order inversa

$ svn log -r 8       # mostra o log para a revisão 8
```

Você também pode examinar o histórico de logs de um único arquivo ou diretório. Por exemplo:

```
$ svn log foo.c
…
$ svn log http://foo.com/svn/trunk/code/foo.c
…
```

Isto vai exibir as mensagens de log *apenas* para aquelas revisões nas quais o arquivo de trabalho (ou a URL) mudaram.

Se você quiser ainda mais informação sobre um arquivo ou diretório, o **svn log** também aceita uma opção `--verbose (-v)`. Como o Subversion lhe permite mover e copiar arquivos e diretórios, é importante ser capaz de rastrear alterações de caminhos no sistema de arquivos, então no modo verboso, o **svn log** vai incluir na sua saída uma lista dos caminhos alterados em uma revisão:

```
$ svn log -r 8 -v
------------------------------------------------------------------------
r8 | sally | 2002-07-14 08:15:29 -0500 | 1 line
Changed paths:
M /trunk/code/foo.c
M /trunk/code/bar.h
A /trunk/code/doc/README

Frozzled the sub-space winch.

------------------------------------------------------------------------
```

O **svn log** também aceita uma opção `--quiet` (`-q`), que suprime o corpo da mensagem de log. Quando combinada com a opção `--verbose`, ela apresentará apenas os nomes dos arquivos mudados.

Por Que o svn log Me Deu Uma Resposta em Branco?

Após trabalhar um pouco com o Subversion, muitos usuários vão se deparar com algo como:

```
$ svn log -r 2
------------------------------------------------------------------------
$
```

À primeira vista, isto parece ser um erro. Mas lembre-se de que as revisões se aplicam a todo o repositório, ao passo que o **svn log** atua em um caminho no repositório. Se você não informar um caminho, o Subversion usa o diretório atual como argumento padrão. Como resultado, se você estiver em um subdiretório de sua cópia de trabalho e tentar ver o log de uma revisão em que nem o diretório nem qualquer item nele contido mudou, o Subversion vai lhe mostrar um log vazio. Se você quiser ver o que mudou naquela revisão, experimente chamar o **svn log** diretamente na URL de mais alto nível de seu repositório, como em **svn log -r 2 http://svn.collab.net/repos/svn**.

Examinando os detalhes das alterações históricas

Nós já vimos o **svn diff** antes—ele exibe as diferenças de arquivo no formato diff unificado; ele foi usado para mostrar as modificações locais feitas em nossa cópia de trabalho antes de serem registradas no repositório.

Na realidade, tem-se que existem *três* usos distintos para o **svn diff**:

- Examinar alterações locais
- Comparar sua cópia de trabalho com o repositório
- Comparar o repositório com o repositório

Examinando Alterações Locais

Como já vimos, executar **svn diff** sem opções vai resultar numa comparação de seus arquivos de trabalho com as cópias “intactas” na área `.svn`:

```
$ svn diff
Index: rules.txt
===================================================================
--- rules.txt (revision 3)
+++ rules.txt (working copy)
@@ -1,4 +1,5 @@
 Be kind to others
 Freedom = Responsibility
 Everything in moderation
-Chew with your mouth open
+Chew with your mouth closed
+Listen when others are speaking
$
```

Comparando a Cópia de Trabalho com o Repositório

Se um único número de revisão for passado para a opção `--revision` (`-r`), então sua cópia de trabalho será comparada com a revisão especificada no repositório.

```
$ svn diff -r 3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 3)
+++ rules.txt (working copy)
@@ -1,4 +1,5 @@
 Be kind to others
 Freedom = Responsibility
 Everything in moderation
-Chew with your mouth open
+Chew with your mouth closed
+Listen when others are speaking
$
```

Comparando o Repositório com o Repositório

Se dois números de revisão, separados por dois-pontos, forem informados em `--revision (-r)`, então as duas revisões serão comparadas diretamente.

```
$ svn diff -r 2:3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 2)
+++ rules.txt (revision 3)
@@ -1,4 +1,4 @@
 Be kind to others
-Freedom = Chocolate Ice Cream
+Freedom = Responsibility
 Everything in moderation
 Chew with your mouth open
$
```

Uma forma mais conveniente de se comparar uma revisão com sua anterior é usando `--change (-c)`:

```
$ svn diff -c 3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 2)
+++ rules.txt (revision 3)
@@ -1,4 +1,4 @@
 Be kind to others
-Freedom = Chocolate Ice Cream
+Freedom = Responsibility
 Everything in moderation
 Chew with your mouth open
$
```

Por último, você pode comparar revisões no repositório até quando você não tem uma cópia de trabalho em sua máquina local, apenas incluindo a URL apropriada na linha de comando:

```
$ svn diff -c 5 http://svn.example.com/repos/example/trunk/text/rules.txt
…
$
```

Navegando pelo repositório

Usando **svn cat** e **svn list**, você pode ver várias revisões de arquivos e diretórios sem precisar mexer nas alterações que estiver fazendo em sua cópia de trabalho. De fato, você nem mesmo precisa de uma cópia de trabalho para usar esses comandos.

svn cat

Se você quiser visualizar uma versão antiga de um arquivo e não necessariamente as diferenças entre dois arquivos, você pode usar o **svn cat**:

```
$ svn cat -r 2 rules.txt
Be kind to others
Freedom = Chocolate Ice Cream
Everything in moderation
Chew with your mouth open
$
```

Você também pode redirecionar a saída diretamente para um arquivo:

```
$ svn cat -r 2 rules.txt > rules.txt.v2
$
```

svn list

O comando **svn list** lhe mostra que arquivos estão no repositório atualmente sem carregá-los para sua máquina local:

```
$ svn list http://svn.collab.net/repos/svn
README
branches/
clients/
tags/
trunk/
```

Se você preferir uma listagem mais detalhada, inclua a opção `--verbose (-v)` para ter uma saída desta forma:

```
$ svn list -v http://svn.collab.net/repos/svn
  20620 harry            1084 Jul 13  2006 README
  23339 harry                 Feb 04 01:40 branches/
  21282 sally                 Aug 27 09:41 developer-resources/
  23198 harry                 Jan 23 17:17 tags/
  23351 sally                 Feb 05 13:26 trunk/
```

As colunas lhe dizem a revisão na qual o arquivo ou diretório sofreu a última (mais recente) modificação, o usuário que o modificou, seu tamanho, se for um arquivo, a data desta última modificação, e o nome do item.

Um **svn list** sem argumentos vai se referir à *URL do repositório* associada ao diretório atual da cópia de trabalho, e *não* ao diretório local da cópia de trabalho. Afinal, se você quer listar o conteúdo de seu diretório local, você pode usar um simples **ls** (ou comando equivalente em seu sistema não-Unix).

Retornando o repositório a momentos antigos

Além de todos os comandos acima, você pode usar o **svn update** e o **svn checkout** com a opção `--revision` para fazer com que toda a sua cópia de trabalho "volte no tempo" [6]:

```
$ svn checkout -r 1729 # Obtém uma nova cópia de trabalho em r1729
…
$ svn update -r 1729 # Atualiza a cópia de trabalho existente para r1729
…
```

Muitos novatos no Subversion tentam usar o **svn update** deste exemplo para "desfazer" modificações submetidas, mas isso não funciona já que você não pode não pode submeter alterações que você obteve voltando uma cópia de trabalho se seus arquivos modificados tiverem novas revisões. Veja "Ressucitando Itens Excluídos" para uma descrição de como "desfazer" um commit.

Finalmente, se você estiver criando uma versão final e quiser empacotar seus arquivos do Subversion mas não gostaria de incluir os incômodos diretórios `.svn` de forma nenhuma, então você pode usar o comando **svn export** para criar uma cópia local de todo o conteúdo de seu repositório mas sem os

[6]Viu? Nós dissemos que o Subversion era uma máquina do tempo.

diretórios `.svn`. Da mesma forma que com o **svn update** e **svn checkout**, você também pode incluir a opção `--revision` ao **svn export**:

```
$ svn export http://svn.example.com/svn/repos1 # Exports latest revision
…
$ svn export http://svn.example.com/svn/repos1 -r 1729
# Exports revision r1729
…
```

Às Vezes Você Só Precisa Limpar

Quando o Subversion modifica sua cópia de trabalho (ou qualquer informação dentro de `.svn`), ele tenta fazer isso da forma mais segura possível. Antes de modificar a cópia de trabalho, o Subversion escreve suas pretensões em um arquivo de log. Depois ele executa os comandos no arquivo de log para aplicar a alteração requisitada, mantendo uma trava na parte relevante da cópia de trabalho enquanto trabalha—para evitar que outros clientes Subversion acessem a cópia de trabalho nesse meio-tempo. Por fim, o Subversion remove o arquivo de log. Arquiteturalmente, isto é similar a um sistema de arquivos com journaling. Se uma operação do Subversion é interrompida (se o processo for morto, ou se a máquina travar, por exemplo), o arquivo de log permanece no disco. Executando novamente os arquivos de log, o Subversion pode completar a operação previamente iniciadas, e sua cópia de trabalho pode manter-se de novo em um estado consistente.

E isto é exatamente o que o **svn cleanup** faz: varre sua cópia de trabalho e executa quaisquer arquivos de log que tenham ficado, removendo as travas da cópia de trabalho durante o processo. Se acaso o Subversion lhe disser que alguma parte de sua cópia de trabalho está “travada” (*locked*), então este é o comando que você deverá rodar. E ainda, o **svn status** mostrará um `L` próximo dos itens que estiverem travados:

```
$ svn status
  L    somedir
M      somedir/foo.c

$ svn cleanup
$ svn status
M      somedir/foo.c
```

Não confunda estas travas da cópia de trabalho com as travas ordinárias que os usuários do Subversion criam ao usar o modelo de controle de versão “travar-modificar-destravar”; veja Os três significados de trava para mais esclarecimentos.

Sumário

Agora nós cobrimos a maioria dos comandos do cliente Subversion. As exceções notáveis são os comandos relacionados com ramificação e mesclagem (veja Capítulo 4, *Fundir e Ramificar*) e propriedades (veja “Propriedades”). Entretanto, você pode querer tirar um momento para folhear Capítulo 9, *Referência Completa do Subversion* para ter uma idéia de todos os muitos comandos diferentes que o Subversion possui—e como você pode usá-los para tornar seu trabalho mais fácil.

Capítulo 3. Tópicos Avançados

Se você está lendo este livro capítulo por capítulo, do inicio ao fim, você deve agora ter adquirido conhecimentos suficientes para usar o cliente Subversion para executar a operações de controle de verão mais comuns. Você entendeu como obter uma cópia de trabalho de um repositório Subversion. Você sente-se confortável para submeter e receber mudanças usando as funções **svn commit** e **svn update**. Você provavelmente desenvolveu um reflexo que lhe impele a executar o comando **svn status** quase inconscientemente. Para todos os intentos e propósitos, você está pronto para usar o Subversion em um ambiente típico.

Mas o conjunto de recursos do Subversion não para nas "operações de controle de versão comuns". Ele tem outras pequenas funcionalidades além de comunicar mudanças de arquivos e diretórios para e a partir de um repositório central.

Este capítulo destaca alguns dos recursos do Subversion que, apesar de importantes, não fazem pare da rotina diária de um usuário típico. Ele assume que você está familiarizado com capacidades básicas de controle de versão sobre arquivos e diretórios. Se não está, você vai querer ler primeiro o Capítulo 1, *Conceitos Fundamentais* e Capítulo 2, *Uso Básico*. Uma vez que você tenha dominado estes fundamentos e terminado este capítulo, você será um usuário avançado do Subversion!

Especificadores de Revisão

Como você viu em "Revisões", números de revisão no Subversion são bastante simples—números inteiros que aumentam conforme você submete mais alterações em seus dados versionados. Assim, não demora muito para que você não se lembre mais do que aconteceu exatamente em toda uma dada revisão. Felizmente, o típico ciclo de trabalho no Subversion frequentemente não precisa que você informe números de revisão arbitrários para as operações que você executa no Subversion. Para aquelas operações que *precisam* de um especificador de revisão, você geralmente informa um número de revisão que você viu em um e-mail da submissão (e-mail de *commit*), na saída de alguma outra operação do Subversion, ou em algum outro contexto que poderia fazer sentido para aquele número em particular.

Mas ocasionalmente, você precisa de um marco de um momento no tempo para o qual você não tem ainda um número de revisão memorizado ou em mãos. Neste caso, além de números inteiros de revisão, o **svn** permite como entrada algumas formas de especificadores de revisão—*termos de revisão*, e datas de revisão.

As várias formas de especificadores de revisão do Subversion podem ser misturadas e correspondidas quando usadas para especificar intervalos de revisão. Por exemplo, você pode usar `-r` *`REV1`*`:`*`REV2`* onde *`REV1`* seja um termo de revisão e *`REV2`* seja um número de revisão, onde *`REV1`* seja uma data e *`REV2`*, um termo de revisão, e por aí vai. Especificadores de revisão individuais são avaliados independentemente, então você pode pôr o que bem quiser junto dos dois-pontos.

Termos de Revisão

O cliente Subversion entende um conjunto de termos de revisão. Estes termos podem ser usados no lugar dos argumentos inteiros para a opção `--revision (-r)`, e são resolvidos para números de revisão específicos pelo Subversion:

HEAD
: A última (ou "mais recente") revisão no repositório.

BASE
: O número de revisão de um item em uma cópia de trabalho. Se o item tiver sido modificado localmente, a "versão BASE" refere-se à forma como o item estaria sem estas modificações locais.

COMMITTED
A revisão mais recente anterior, ou igual a, BASE, na qual o item foi modificado.

PREV
A revisão imediatamente *anterior* à última revisão na qual o item foi modificado. Tecnicamente, isto se resume a COMMITTED-1.

Como pode ser deduzido de suas descrições, os termos de revisão PREV, BASE, e COMMITTED são usados apenas quando se referirem a um caminho numa cópia de trabalho—eles não se aplicam a URLs do repositório. HEAD, por outro lado, pode ser usado em conjunto para qualquer um desses tipos de caminho.

Aqui estão alguns exemplos da utilização de termos de revisão:

```
$ svn diff -r PREV:COMMITTED foo.c
# exibe a última alteração submetida em foo.c

$ svn log -r HEAD
# mostra a mensagem de log do último registro no repositório

$ svn diff -r HEAD
# compara sua cópia de trabalho (com todas suas alterações locais) com a
# última versão na árvore do diretório

$ svn diff -r BASE:HEAD foo.c
# compara a versão inalterada de foo.c com a última versão de
# foo.c no repositório

$ svn log -r BASE:HEAD
# mostra todos os logs das submissões para o diretório atual versionado
# desde a última atualização que você fez em sua cópia de trabalho

$ svn update -r PREV foo.c
# retorna à última alteração feita em foo.c, reduzindo o número de
# revisão de foo.c

$ svn diff -r BASE:14 foo.c
# compara a versão inalterada de foo.c com o conteúdo que foo.c tinha na
# revisão 14
```

Datas de Revisão

Números de revisão não revelam nada sobre o mundo fora do sistema de controle de versão, mas algumas vezes você precisa correlacionar um momento em tempo real com um momento no histórico de revisões. Para facilitar isto, a opção `--revision (-r)` também pode aceitar especificadores de data delimitados por chaves ({ e }) como entrada. O Subversion aceita datas e horas no padrão ISO-8601, além de alguns poucos outros. Aqui estão alguns exemplos. (Lembre-se de usar aspas para delimitar quaisquer datas que contenham espaços.)

```
$ svn checkout -r {2006-02-17}
$ svn checkout -r {15:30}
$ svn checkout -r {15:30:00.200000}
$ svn checkout -r {"2006-02-17 15:30"}
$ svn checkout -r {"2006-02-17 15:30 +0230"}
$ svn checkout -r {2006-02-17T15:30}
$ svn checkout -r {2006-02-17T15:30Z}
```

```
$ svn checkout -r {2006-02-17T15:30-04:00}
$ svn checkout -r {20060217T1530}
$ svn checkout -r {20060217T1530Z}
$ svn checkout -r {20060217T1530-0500}
…
```

Quando você especifica uma data, o Subversion resolve aquela data para a revisão mais recente do repositório com aquela data, e então continua a operação usando o número de revisão obtido:

```
$ svn log -r {2006-11-28}
------------------------------------------------------------------------
r12 | ira | 2006-11-27 12:31:51 -0600 (Mon, 27 Nov 2006) | 6 lines
…
```

O Subversion está um dia adiantado?

Se você especificar uma única data como uma revisão sem especificar uma hora do dia (por exemplo `2006-11-27`), você pode pensar que o Subversion deveria dar a você a última revisão que tivesse ocorrido em 27 de novembro. Entretanto, você vai obter uma revisão do dia 26, ou mesmo anterior a isso. Lembre-se de que o Subversion vai procurar a *revisão do repositório mais recente* que a da data que você informou. Se você informar uma data sem a parte de horário, como `2006-11-27`, o Subversion assume um horário de 00:00:00, então procurar pela revisão mais recente não vai retornar nada do dia 27.

Se você quiser incluir o dia 27 em sua busca, você pode tanto especificar o dia 27 com o horário (`{"2006-11-27 23:59"}`), ou apenas especificar o próximo dia (`{2006-11-28}`).

Você também pode usar intervalos de datas. O Subversion vai encontrar todas as revisões entre as datas, inclusive:

```
$ svn log -r {2006-11-20}:{2006-11-29}
…
```

Uma vez que a data e horário (*timestamp*) de uma revisão é armazenada como uma propriedade da revisão não-versionada e passível de alteração (veja “Propriedades”, essas informações de data e horário podem ser modificadas para representar falsificações completas da cronologia real, ou mesmo podem ser removidas inteiramente. A capacidade do Subversion de converter corretamente datas de revisão em números de revisão depende da manutenção da ordem seqüencial desta informação temporal—quanto mais recente uma revisão, mais recente é sua informação de data e horário. Se esta ordenação não for mantida, você perceberá que tentar usar datas para especificar intervalos de revisão em seu repositório nem sempre retornará os dados que você espera.

Propriedades

Nós já abordamos em detalhes como o Subversion armazena e recupera várias versões de arquivos e diretórios em seu repositório. Capítulos inteiros têm sido focados nesta parte mais fundamental das funcionalidades providas pela ferramenta. E se o suporte a versionamento parar por aí, o Subversion ainda seria completo do ponto de vista do controle de versão.

Mas não pára por aí.

Adicionalmente ao versionamento de seus arquivos e diretórios, o Subversion permite interfaces para adição, modificação e remoção de metadados versionados em cada um de seus arquivos e diretórios sob controle de versão. Chamamos estes metadados de *propriedades*, e eles podem ser entendidos

como tabelas de duas colunas que mapeiam nomes de propriedades a valores arbitrários anexados a cada item em sua cópia de trabalho. Falando de uma forma geral, os nomes e valores das propriedades podem ser quaisquer coisas que você queira, com a restrição de que os nomes devem ser texto legível por humanos. E a melhor parte sobre estas propriedades é que elas, também, são versionadas, tal como o conteúdo textual de seus arquivos. Você pode modificar, submeter, e reverter alterações em propriedades tão facilmente como em qualquer alteração no conteúdo de arquivos. E o envio e recebimento das modificações em propriedades ocorrem como parte de suas operações de submissão (*commit*) e atualização (*update*)—você não tem que mudar seus procedimentos básicos para utilizá-los.

O Subversion reservou um conjunto de propriedades cujos nomes começam com `svn:` para si próprio. Ainda que haja apenas um conjunto útil de tais propriedades em uso hoje em dia, você deve evitar criar suas propriedades específicas com nomes que comecem com este prefixo. Do contrário, você corre o risco de que uma versão futura do Subversion aumente seu suporte a recursos ou comportamentos a partir de uma propriedade de mesmo nome, mas talvez com um significado completamente diferente.

Propriedades aparecem em qualquer parte no Subversion, também. Da mesma maneira que arquivos e diretórios podem ter nomes de propriedades arbitrários e valores anexados a eles, cada revisão como um todo pode ter propriedades arbitrárias anexadas a si própria. As mesmas restrições se aplicam;mdash;nomes que sejam legíveis por humanos e valores com qualquer coisa que você queira, inclusive dados binários. A principal diferença é que as propriedades de uma revisão não são versionadas. Em outras palavras, se você modificar o valor de, ou excluir uma propriedade de uma revisão, não há uma forma de recuperar seu valor anterior no Subversion.

O Subversion não tem nenhuma política em particular em relação ao uso de propriedades. Ele apenas solicita que você não use nomes de propriedades que comecem com o prefixo `svn:`. Este é o espaço de nomes (*namespace*) que ele reserva para uso próprio. E o Subversion, de fato, faz uso de propriedades, tanto do tipo versionadas quando das não-versionadas. Certas propriedades versionadas têm um significado especial ou certos efeitos quando encontradas em arquivos e diretórios, ou carregam alguma pequena informação sobre a revisão na qual estes se encontram. Certas propriedades de revisão são automaticamente anexadas às revisões pelo processo de submissão de alterações do Subversion, e obtém informação sobre a revisão. Muitas dessas propriedades são mencionadas em algum lugar neste ou em outros capítulos como parte de tópicos mais gerais aos quais estão relacionadas. Para uma lista exaustiva das propriedades pré-definidas do Subversion, veja “Subversion properties”.

Nesta seção, vamos examinar a utilidade—tanto para os usuários quanto para o próprio Subversion —do suporte a propriedades. Você vai aprender sobre os subcomandos de **svn** que lidam com propriedades, e como alterações nas propriedades afetam seu fluxo de trabalho normal no Subversion.

Por que Propriedades?

Como o Subversion usa propriedades para armazenar informações extras sobre arquivos, diretórios, e revisões que as contém, você também pode usar propriedades de forma semelhante. Você poderia achar útil ter um lugar próximo de seus dados versionados para pendurar metadados personalizados sobre estes dados.

Digamos que você quer elaborar um website que armazene muitas fotos digitais, e as exiba com legendas e a data e hora em que foram tiradas. Só que seu conjunto de fotos está mudando constantemente, e você gostaria de automatizar este site tanto quanto possível. Estas fotos podem ser um pouco grandes, então, como muito comum em sites desse tipo, você quer disponibilizar prévias em miniatura de suas fotos para os visitantes de seu site.

Agora, você pode ter esta funcionalidade usando arquivos tradicionais. Isto é, você pode ter seus arquivos `image123.jpg` e `image123-thumbnail.jpg` lado a lado em um diretório. Ou se você quiser manter os mesmos nomes de arquivos, você poderia ter suas miniaturas em um diretório diferente, como `thumbnails/image123.jpg`. Você também pode armazenar suas legendas e datas e horários de forma parecida, também separadas do arquivo da foto original. Mas o problema aqui é que o conjunto de arquivos aumenta aos múltiplos para cada nova foto adicionada ao site.

Agora considere o mesmo website desenvolvido de forma a fazer uso das propriedades de arquivo do Subversion. Imagine ter um único arquivo de imagem, `image123.jpg`, e então propriedades definidas neste arquivo chamadas `caption`, `datestamp`, ou mesmo `thumbnail` (respectivamente para a legenda, data e hora, e miniatura da imagem). Agora sua cópia de trabalho parece muito mais gerenciável—de fato, numa primeira visualização, não parece haver nada mais além dos arquivos de imagem lá dentro. Mas seus scripts de automação sabem mais. Eles sabes que podem usar o **svn** (ou melhor ainda, eles podem usar a linguagem incorporada ao Subversion—veja “Usando as APIs”) para para extrair as informações extras que seu site precisa exibir sem ter que lê-las de um arquivo de índices nem precisar de preocupar em estar manipulando caminhos dos arquivos.

As propriedades personalizadas de revisões também são freqüentemente usadas. Um de seus usos comuns é uma propriedades cujo valor contém um recurso de ID de rastreamento ao qual a revisão está associada, talvez pelo fato de a alteração feita nesta revisão corrigir um problema relatado externamente com aquele ID. Outros usos incluem a inserção de nomes mais amigáveis à revisão—pode ser difícil lembrar que a revisão 1935 foi uma revisão testada completamente. Mas se houver, digamos, uma propriedade `resultado-dos-testes` nesta revisão com o valor `todos passaram`, esta á uma informação bem mais significativa de se ter.

Procurabilidade (ou, Porque *Não* Usar Propriedades)

Para algo com tamanha , as propriedades do Subversion—ou, mais precisamente, as interfaces disponíveis para elas—têm uma grande falha: apesar de ser simples *criar* uma propriedade específica, *procurar* esta propriedade posteriormente são outros quinhentos.

Tentar localizar uma propriedade específica de uma revisão geralmente envolve executar uma busca linear ao longo de todas as revisões do repositório, perguntando para cada revisão, "Você tem a propriedade que eu estou procurando?" Tentar encontrar uma propriedade versionada é complicado, também, e quase sempre envolve um recursivo **svn propget** ao longo de toda a cópia de trabalho. Esta situação pode até não ser tão ruim quando uma busca linear por todas as revisões. Mas certamente deixa muito a desejar tanto em termos de performance quanto mesmo da probabilidade de sucesso, especialmente se o escopo de sua busca envolver uma cópia de trabalho da raiz de seu repositório.

Por este motivo, você deve escolher—especialmente no caso de propriedades de revisões—entre simplesmente adicionar seus metadados na mensagem de log da revisão, usando algum padrão de formatação (talvez até a partir de ferramentas de programação) que permita uma filtragem rápida a partir da saída do comando **svn log**. É bastante comum de se ver mensagens de log do Subversion parecidas com:

```
Issue(s): IZ2376, IZ1919
Reviewed by:  sally

Isto corrige um erro de falha de segmentação no gerenciador de processos
…
```

Mas isto resulta em outra dificuldade. O Subversion não provê ainda uma mecanismo de modelos para mensagens de log, que poderia ajudar bastante os usuários a manter consistentemente o formato de seus metadados incluídos em suas mensagens de log.

Manipulando Propriedades

O comando **svn** oferece algumas poucas maneiras de se adicionar ou modificar propriedades de arquivos e diretórios. Para propriedades com valores pequenos, legíveis por humanos, talvez a forma mais simples de se adicionar uma nova propriedade é especificar o nome e o valor da propriedade na linha de comando com o subcomando **propset**.

```
$ svn propset copyright '(c) 2006 Red-Bean Software' calc/button.c
property 'copyright' set on 'calc/button.c'
$
```

Mas sempre podemos contar com a flexibilidade que o Subversion oferece para seus valores de propriedades. E se você está planejando ter texto com múltiplas linhas, ou mesmo valores binários para o valor da propriedade, você provavelmente não vai informar este valor pela linha de comando. Então, o subcomando **propset** leva uma opção `--file (-F)` para especificar o nome de um arquivo que contém o valor para a nova propriedade.

```
$ svn propset license -F /path/to/LICENSE calc/button.c
property 'license' set on 'calc/button.c'
$
```

Há algumas restrições sobre os nomes que você pode usar para propriedades. Um nome de propriedade deve começar com uma letra, dois-pontos (`:`), ou um caractere sublinha (`_`); e depois disso, você também pode usar dígitos, hifens (`-`), e pontos (`.`). [1]

Além do **propset**, o programa **svn** também oferece comando **propedit**. Este comando usa o editor de texto configurado (veja "Configuração") para adicionar ou modificar propriedades. Quando você executa este comando, o **svn** chama seu programa editor em um arquivo temporário que contém o valor atual da propriedade (ou o qual é vazio, se você estiver adicionando uma nova propriedade). Então, você apenas modifica esse valor em seu editor até que ele represente o novo valor que você quer armazenar para a propriedade. Se o Subversion identificar que no momento você está modificando o valor da propriedade existente, ele irá aceitá-lo como novo valor da propriedade. Se você sair de seu editor sem fazer qualquer alteração, nenhuma modificação irá ocorrer:

```
$ svn propedit copyright calc/button.c  ### sai do editor sem fazer nada
No changes to property 'copyright' on 'calc/button.c'
$
```

Devemos notar que, como qualquer outro subcomando do **svn**, estes que são relacionados a propriedades podem agir em diversos caminhos de uma só vez. Isto lhe permite modificar propriedades em todo um conjunto de arquivos com um único comando. Por exemplo, nós poderíamos ter feito:

```
$ svn propset copyright '(c) 2006 Red-Bean Software' calc/*
property 'copyright' set on 'calc/Makefile'
property 'copyright' set on 'calc/button.c'
property 'copyright' set on 'calc/integer.c'
…
$
```

Toda esta adição e alteração de propriedades não é realmente muito útil se você não puder obter facilmente o valor armazenado da propriedade. Então o programa **svn** dispõe de dois subcomando para exibição dos nomes e valores das propriedades armazenadas nos arquivos e diretórios. O comando **svn proplist** vai listar os nomes das propriedades que existem naquele caminho. Uma vez que você saiba os nomes das propriedades do nó, você pode verificar seus valores individualmente usando **svn propget**. Este comando mostrará, dado um nome de propriedade e um caminho (ou conjunto de caminhos), o valor da propriedade para a saída padrão.

```
$ svn proplist calc/button.c
Properties on 'calc/button.c':
```

[1] Se você é familiarizado com XML, este é exatamente o subconjunto ASCII da sintaxe de um "Nome" XML.

```
  copyright
  license
$ svn propget copyright calc/button.c
(c) 2006 Red-Bean Software
```

Há ainda uma variação do comando **proplist** que lista tanto o nome quanto o valor de todas as propriedades. Apenas informe a opção `--verbose` (`-v`).

```
$ svn proplist -v calc/button.c
Properties on 'calc/button.c':
  copyright : (c) 2006 Red-Bean Software
  license : ================================================================
Copyright (c) 2006 Red-Bean Software.  All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions, and the recipe for Fitz's famous
red-beans-and-rice.
…
```

O último subcomando relacionado a propriedades é o **propdel**. Como o Subversion permite armazenar propriedades com valores vazios, você não pode remover uma propriedade usando **propedit** ou **propset**. Por exemplo, este comando *não* vai surtir o efeito desejado:

```
$ svn propset license '' calc/button.c
property 'license' set on 'calc/button.c'
$ svn proplist -v calc/button.c
Properties on 'calc/button.c':
  copyright : (c) 2006 Red-Bean Software
  license :
$
```

Você precisa usar o subcomando **propdel** para remover propriedades completamente. A sintaxe é semelhante a dos outros comandos de propriedades:

```
$ svn propdel license calc/button.c
property 'license' deleted from 'calc/button.c'.
$ svn proplist -v calc/button.c
Properties on 'calc/button.c':
  copyright : (c) 2006 Red-Bean Software
$
```

Lembra das propriedades não-versionadas de revisões? Você pode modificá-las, também, usando os mesmo subcomandos do **svn** que acabamos de descrever. Apenas adicione o parâmetro `--revprop` na linha de comando, e especifique a revisão cujas propriedades você quer modificar. Como as revisões são globais, você não precisa especificar um caminho para para estes comandos relacionados a propriedades enquanto estiver em uma cópia de trabalho do repositório cuja propriedade de revisão você queira alterar. Por outro lado, você pode apenas especificar a URL de qualquer caminho de seu interesse no repositório (incluindo a URL raiz do repositório). Por exemplo, você pode querer trocar a mensagem de log de um registro de alteração de uma revisão existente. [2] Se seu diretório atual

[2]Correção de erros de ortografia, gramaticais “outros ajustes simples de texto” nas mensagens de log de uma registro talvez seja o uso mais comum da opção `--revprop` option.

for parte da cópia de trabalho de seu repositório, você pode simplesmente executar o comando **svn propset** sem nenhum caminho:

```
$ svn propset svn:log '* button.c: Fix a compiler warning.' -r11 --revprop
property 'svn:log' set on repository revision '11'
$
```

Mas mesmo que você não tenha criado uma cópia de trabalho a partir do repositório, você ainda assim pode proceder com modificação de propriedades informando a URL raiz do repositório:

```
$ svn propset svn:log '* button.c: Fix a compiler warning.' -r11 --revprop \
              http://svn.example.com/repos/project
property 'svn:log' set on repository revision '11'
$
```

Perceba que a permissão para se alterar estas propriedades não-versionadas deve ser explicitamente concedida pelo administrador do repositório (veja "Commit Log Message Correction"). Isto é porque as propriedades não são versionadas, então você corre o risco de perder informação se não for cuidadoso com suas alterações. O administrador do repositório pode definir formas de proteção contra perda de dados, e por padrão, além de que as modificações de propriedades não-versionadas são desabilitadas por padrão.

Usuários poderia, quando possível, usar **svn propedit** ao invés de **svn propset**. Ainda que o resultado da execução dos comandos seja o mesmo, o primeiro vai lhes permitir visualizar o valor atual da propriedade que querem modificar, o que ajuda a conferir que estão, de fato, fazer a alteração que acham que estão fazendo. Isto é especialmente verdadeiro ao modificar as propriedades não-versionadas de revisão. E ainda, é significativamente modificar valores de propriedades em múltiplas linhas em um editor de texto do que pela linha de comando.

Propriedades e o Fluxo de Trabalho no Subversion

Agora que você está familiarizado com todos os subcomandos **svn** relacionados a propriedades, vamos ver como as modificações de propriedades afetam o fluxo de trabalho usual do Subversion. Como mencionado anteriormente, as propriedades de arquivos e diretórios são versionadas, tal como os conteúdos de arquivos. Como resultado, o Subversion dispõe dos mesmos recursos para mesclar—de forma limpa ou com conflitos—alterações de terceiros às nossas próprias.

E como com conteúdos de arquivos, suas mudanças de propriedades são modificações locais, que são tornadas permanentes apenas quando submetidas ao repositório com o comando **svn commit**. Suas alterações de propriedades também podem ser facilmente desfeitas—o comando **svn revert** vai restaurar seus arquivos e diretórios para seus estados inalterados—conteúdos, propriedades, e tudo. Também, você pode obter informações interessantes sobre o estado de suas propriedades de arquivos e diretórios usando os comandos **svn status** e **svn diff**.

```
$ svn status calc/button.c
 M     calc/button.c
$ svn diff calc/button.c
Property changes on: calc/button.c
___________________________________________________________________
Name: copyright
   + (c) 2006 Red-Bean Software

$
```

Note como o subcomando **status** exibe `M` na segunda coluna ao invés de na primeira. Isto é porque modificamos as propriedades de `calc/button.c`, mas não seu conteúdo textual. Se tivéssemos

modificado ambos, deveríamos ver um `M` na primeira coluna também (veja "Obtendo uma visão geral de suas alterações").

Conflitos de Propriedades

Da mesma forma que para conteúdo de arquivos, modificações de propriedades podem conflitar com alterações submetidas por outros. Se você atualizar sua cópia de trabalho e receber alterações de propriedade em um objeto versionado que vão de encontro às suas, o Subversion vai informar que o objeto está em um estado conflituoso.

```
% svn update calc
M  calc/Makefile.in
 C calc/button.c
Updated to revision 143.
$
```

O Subversion também vai criar, no mesmo diretório que o objeto em conflito, um arquivo com a extensão `.prej` contendo os detalhes do conflito. Você deve examinar o conteúdo deste arquivo para decidir como resolver o conflito. Até que o conflito seja resolvido, você verá um `C` na segunda coluna da saída do **svn status** para o objeto em questão, e as tentativas de submeter suas modificações locais irão falhar.

```
$ svn status calc
 C     calc/button.c
?      calc/button.c.prej
$ cat calc/button.c.prej
prop 'linecount': user set to '1256', but update set to '1301'.
$
```

Para resolver conflitos de propriedades, apenas garanta que as propriedades conflituosas contenham os valores que deveriam conter, e então use o comando **svn resolved** para informar ao Subversion que você resolveu o problema manualmente.

Você também deve ter notado a forma não-padrão como o Subversion atualmente exibe diferenças de propriedades. Você ainda pode executar **svn diff** e redirecionar a saída para criar um arquivo de patch usável. O programa **patch** vai ignorar patches de propriedades—como regra, ele ignora qualquer coisa que não consiga entender. Isso significa, infelizmente, que para aplicar completamente um patch gerado pelo **svn diff**, quaisquer alterações em propriedades precisarão ser aplicadas manualmente.

Definição Automática de Propriedades

Propriedades são um poderoso recurso do Subversion, agindo como componentes chave em muitos outros recursos apresentados neste e em outros capítulos—suporte a diferenciação e mesclagem textual, substituição de palavras-chave, conversão de delimitadores de linha, etc. Mas para aproveitar plenamente as propriedades, elas devem ser definidas nos arquivos e diretórios certos. Infelizmente, este passo é facilmente esquecido na rotina de ações de usuário, especialmente pelo fato de que a não atribuição de uma propriedade normalmente não resulta em nenhum erro óbvio (ao menos quando comparado a, digamos, esquecer de adicionar um arquivo ao controle de versão). Para ajudar que suas propriedades sejam aplicadas em seus devidos locais, o Subversion dispõem de uma porção de recursos simples e poderosos.

Sempre que você inclui um arquivo ao controle de versão usando os comando **svn add** ou **svn import**, o Subversion tenta ajudar criando algumas propriedades de arquivo automaticamente. Primeiramente, em sistema operacionais cujos sistemas de arquivo suportem bits de permissão de execução,

o Subversion automaticamente vai definir a propriedade `svn:executable` nos arquivos recém adicionados ou importados nos quais o bit de execução esteja ativo. (Veja “Executabilidade de Arquivo” para mais sobre esta propriedade.) Em segundo lugar, ele executa uma heurística bem básica para identificar se o arquivo contém algum conteúdo que seja legível por humanos. Se não, o Subversion automaticamente vai definir a propriedade `svn:mime-type` naquele arquivo para `application/octet-stream` (o tipo MIME genérico para “isto é um conjunto de bytes”). Claro que se o Subversion identificou corretamente, ou se você quiser definir a propriedade `svn:mime-type` para algo mais preciso—talvez `image/png` ou `application/x-shockwave-flash`—você sempre pode remover ou editar esta propriedade. (Para mais sobre como o Subversion faz uso dos tipos MIME, veja “Tipo de Conteúdo do Arquivo”.)

O Subversion também oferece, através de seu sistema de configuração em tempo de execução (veja “Área de Configuração do Tempo de Execução”), um mecanismo mais flexível de definição automática de propriedades que permite a você criar mapeamentos de padrões de nome de arquivos para nomes de propriedades e valores. Novamente, estes mapeamentos afetam adições e importações e não apenas podem sobrescrever a identificação sobre o tipo MIME padrão feita pelo Subversion durante suas operações, como pode definir propriedades do Subversion adicionais e personalizadas também. Por exemplo, você poderia criar um mapeamento que dissesse que sempre que você adicionar arquivos JPEG—aqueles cujos nomes casem com o padrão `*.jpg`—o Subversion deveria automaticamente definir a propriedade `svn:mime-type` destes arquivos para `image/jpeg`. Ou talvez quaisquer arquivos que correspondam ao padrão `*.cpp` deveriam ter a propriedade `svn:eol-style` definida para `native`, e `svn:keywords` atribuída para `Id`. Suporte a propriedades automáticas talvez seja a ferramenta mais prática no conjunto de ferramentas do Subversion. Veja “Configuração” para mais sobre a configuração deste recurso.

Portabilidade de Arquivo

Felizmente, para os usuários do Subversion que rotineiramente se encontram em diferentes computadores, com diferentes sistemas operacionais, o programa de linha de comando do Subversion comporta-se quase que da mesma forma em todos os sistemas. Se você sabe como usar o **svn** em uma plataforma, você saberá como manuseá-lo em qualquer outra.

Entretanto, o mesmo nem sempre é verdade em outras classes de software em geral, ou nos atuais arquivos que você mantém no Subversion. Por exemplo, em uma máquina Windows, a definição de um “arquivo de texto” seria similar à usada em uma máquina Linux, porém com uma diferença chave—os caracteres usados para marcar o fim das linhas destes arquivos. Existem outras diferenças também. As plataformas Unix têm (e o Subversion suporta) links simbólicos; Windows não. As plataformas Unix usam as permissões do sistema de arquivos para determinar a executabilidade; Windows usa as extensões no nome do arquivo.

Pela razão de que o Subversion não está em condição de unir o mundo inteiro em definições comuns e implementações de todas estas coisas, o melhor que podemos fazer é tentar ajudar a tornar sua vida mais simples quando você precisar trabalhar com seus arquivos e diretórios versionados em múltiplos computadores e sistemas operacionais. Esta seção descreve alguns dos meios de como o Subversion faz isto.

Tipo de Conteúdo do Arquivo

O Subversion combina a qualidade das muitas aplicações que reconhecem e fazem uso dos tipos de conteúdo do *Multipurpose Internet Mail Extensions* (MIME). Além de ser um local de armazenamento de propósito geral para um tipo de conteúdo do arquivo, o valor da propriedade de arquivo `svn:mime-type` determina algumas características comportamentais do próprio Subversion.

Identificando Tipos de Arquivo

Vários programas nos sistemas operacionais mais modernos fazem suposições sobre o tipo e formato do conteúdo de um arquivo pelo nome do arquivo, especificamente por sua extensão. Por exemplo, arquivos cujos nomes terminam em `.txt` são, geralmente, supostos ser legíveis por humanos, passíveis de serem compreendidos por simples leitura, em vez dos que requerem processamento complexo para os decifrar. Por outro lado, arquivos cujos nomes terminam em `.png` assume-se serem do tipo *Portable Network Graphics*— que não são legíveis por humanos, sendo perceptíveis apenas quando interpretados pelo software que entende o formato PNG, e pode tornar a informação neste formato como uma imagem desenhada por linhas.

Infelizmente, algumas destas extensões têm seus significados modificados ao longo do tempo. Quando os computadores pessoais apareceram pela primeira vez, um arquivo chamado `README.DOC` certamente era um arquivo de texto simples, como são hoje os arquivos `.txt`. Porém, no meio dos anos de 1990, você poderia apostar que um arquivo com este nome não seria mais um arquivo de texto simples, mas sim um documento do Microsoft Word em um formato proprietário e humanamente ilegível. Mas esta mudança não ocorreu da noite para o dia—houve certamente um período de confusão para os usuário de computador sobre o que exatamente eles tinham em mãos quando viam um arquivo `.DOC`. [3]

A popularidade das redes de computadores lançou ainda mais dúvidas sobre o mapeamento entre um nome de arquivo e seu conteúdo. Com informações sendo servidas através das redes e geradas dinamicamente por scripts no servidor, freqüentemente, observava-se arquivos não reais e, portanto, sem nome. Os servidores Web, por exemplo, precisavam de algum outro modo para dizer aos navegadores que eles estavam baixando um arquivo, assim o navegador poderia fazer algo inteligente com esta informação, quer seja para exibir os dados usando um programa registrado para lidar com este tipo de dados, quer seja para solicitar ao usuário onde armazenar os dados baixados.

Finalmente, um padrão surgiu para, entre outras coisas, descrever o conteúdo de um fluxo de dados. Em 1996, a RFC2045 foi publicada, a primeira de cinco RFC's descrevendo o MIME. Esta RFC descreve o conceito de tipos e subtipos de mídia, e recomenda uma sintaxe para a representação destes tipos. Hoje, os tipos de mídia MIME—ou “tipos MIME”—são usados quase que universalmente em todas as aplicações de e-mail, servidores Web e outros softwares como o mecanismo de fato para esclarecer a confusão do conteúdo de arquivo.

Por exemplo, um dos benefícios que o Subversion tipicamente fornece é a fusão contextual, baseada nas linhas, das mudanças recebidas do servidor durante uma atualização em seu arquivo de trabalho. Mas, para arquivos contendo dados não-textuais, muitas vezes não existe o conceito de “linha”. Assim, para os arquivos versionados cuja propriedade `svn:mime-type` é definida com um tipo MIME não-textual (geralmente, algo que não inicie com `text/`, embora existam exceções), o Subversion não tenta executar fusões contextuais durante as atualizações. Em vez disso, quando você modifica localmente um arquivo binário em sua cópia de trabalho, na momento da atualização, seu arquivo não é mexido, pois o Subversion cria dois novos arquivos. Um deles tem a extensão `.oldrev` e contém a revisão BASE do arquivo. O outro arquivo tem uma extensão `.newrev` e contém o conteúdo da revisão atualizada do arquivo. Este comportamento serve de proteção ao usuário contra falhas na tentativa de executar fusões contextuais nos arquivos que simplesmente não podem ser contextualmente fundidos.

Além disso, se a propriedade `svn:mime-type` estiver definida, então o módulo Apache do Subversion usará seu valor para preencher o cabeçalho HTTP `Content-type:` quando responder a solicitações GET. Isto oferece ao navegador web uma dica crucial sobre como exibir um arquivo quando você o utiliza para examinar o conteúdo de seu repositório Subversion.

[3] Você acha que foi complicado? Durante este mesmo período, o WordPerfect também usou `.DOC` como extensão para seu formato de arquivo proprietário!

Executabilidade de Arquivo

Em muitos sistemas operacionais, a capacidade de executar um arquivo como um comando é comandada pela presença de um bit de permissão para execução. Este bit, usualmente, vem desabilitado por padrão, e deve ser explicitamente habilitado pelo usuário em cada arquivo que seja necessário. Mas seria um grande incômodo ter que lembrar, exatamente, quais arquivos de uma cópia de trabalho verificada recentemente estavam com seus bits de execução habilitados, e, então, ter que trocá-los. Por esta razão, o Subversion oferece a propriedade `svn:executable`, que é um modo de especificar que o bit de execução para o arquivo no qual esta propriedade está definida deve ser habilitado, e o Subversion honra esta solicitação ao popular cópias de trabalho com tais arquivos.

Esta propriedade não tem efeito em sistemas de arquivo que não possuem o conceito de bit de permissão para executável, como, por exemplo, FAT32 e NTFS. [4] Além disso, quando não houver valor definido, o Subversion forçará o valor `*` ao definir esta propriedade. Por fim, esta propriedade só é válido em arquivos, não em diretórios.

Seqüência de Caracteres de Fim-de-Linha

A não você ser que esteja usando a propriedade `svn:mime-type` em um arquivo sob controle de versão, o Subversion assume que o arquivo contém dados humanamente legíveis. De uma forma geral, o Subversion somente usa esse conhecimento para determinar se os relatórios de diferenças contextuais para este arquivo são possíveis. Ao contrário, para o Subversion, bytes são bytes.

Isto significa que, por padrão, o Subversion não presta qualquer atenção para o tipo de *marcadores de fim-de-linha, ou end-of-line (EOL)* usados em seus arquivos. Infelizmente, diferentes sistemas operacionais possuem diferentes convenções sobre qual seqüência de caracteres representa o fim de uma linha de texto em um arquivo. Por exemplo, a marca usual de término de linha usada por softwares na plataforma Windows é um par de caracteres de controle ASCII—um retorno de carro (`CR`) seguido por um avanço de linha (`LF`). Os softwares em Unix, entretanto, utilizam apenas o caracter `LF` para definir o término de uma linha.

Nem todas as ferramentas nestes sistemas operacionais compreendem arquivos que contêm terminações de linha em um formato que difere do *estilo nativo de terminação de linha* do sistema operacional no qual estão executando. Assim, normalmente, programas Unix tratam o caracter `CR`, presente em arquivos Windows, como um caracter normal (usualmente representado como `^M`), e programas Windows juntam todas as linhas de um arquivo Unix dentro de uma linha enorme, porque nenhuma combinação dos caracteres de retorno de carro e avanço de linha (ou `CRLF`) foi encontrada para determinar os términos das linhas.

Esta sensibilidade quanto aos marcadores EOL pode ser frustrante para pessoas que compartilham um arquivo em diferentes sistemas operacionais. Por exemplo, considere um arquivo de código-fonte, onde desenvolvedores que editam este arquivo em ambos os sistemas, Windows e Unix. Se todos os desenvolvedores sempre usarem ferramentas que preservem o estilo de término de linha do arquivo, nenhum problema ocorrerá.

Mas na prática, muitas ferramentas comuns, ou falham ao ler um arquivo com marcadores EOL externos, ou convertem as terminações de linha do arquivo para o estilo nativo quando o arquivo é salvo. Se o precedente é verdadeiro para um desenvolvedor, ele deve usar um utilitário de conversão externo (tal como **dos2unix** ou seu similar, **unix2dos**) para preparar o arquivo para edição. O caso posterior não requer nenhuma preparação extra. Mas ambos os casos resultam em um arquivo que difere do original literalmente em cada uma das linhas! Antes de submeter suas alterações, o usuário tem duas opções. Ou ele pode utilizar um utilitário de conversão para restaurar o arquivo modificado para o mesmo estilo de término de linha utilizado antes de suas edições serem feitas. Ou ele pode simplesmente submeter o arquivo—as novas marcas EOL e tudo mais.

O resultado de cenários como estes incluem perda de tempo e modificações desnecessárias aos arquivos submetidos. A perda de tempo é suficientemente dolorosa. Mas quando submissões mudam

[4] Os sistemas de arquivos do Windows usam extensões de arquivo (tais como `.EXE`, `.BAT`, e `.COM`) para indicar arquivos executáveis.

cada uma das linhas em um arquivo, isso dificulta o trabalho de determinar quais dessas linhas foram modificadas de uma forma não trivial. Onde o bug foi realmente corrigido? Em qual linha estava o erro de sintaxe introduzido?

A solução para este problema é a propriedade `svn:eol-style`. Quando esta propriedade é definida com um valor válido, o Subversion a utiliza para determinar que tratamento especial realizar sobre o arquivo de modo que o estilo de término de linha do arquivo não fique alternando a cada submissão vinda de um sistema operacional diferente. Os valores válidos são:

`native`
: Isso faz com que o arquivo contenha as marcas EOL que são nativas ao sistema operacional no qual o Subversion foi executado. Em outras palavras, se um usuário em um computador Windows adquire uma cópia de trabalho que contém um arquivo com a propriedade `svn:eol-style` atribuída para `native`, este arquivo conterá `CRLF` como marcador EOL. Um usuário Unix adquirindo uma cópia de trabalho que contém o mesmo arquivo verá `LF` como marcador EOL em sua cópia do arquivo.

  Note que o Subversion na verdade armazenará o arquivo no repositório usando marcadores normalizados como `LF` independentemente do sistema operacional. Isto, no entanto, será essencialmente transparente para o usuário.

`CRLF`
: Isso faz com que o arquivo contenha seqüências `CRLF` como marcadores EOL, independentemente do sistema operacional em uso.

`LF`
: Isso faz com que o arquivo contenha caracteres `LF` como marcadores EOL, independentemente do sistema operacional em uso.

`CR`
: Isso faz com que o arquivo contenha caracteres `CR` como marcadores EOL, independentemente do sistema operacional em uso. Este estilo de término de linha não é muito comum. Ele foi utilizado em antigas plataformas Macintosh (nas quais o Subversion não executa regularmente).

Ignorando Itens Não-Versionados

Em qualquer cópia de trabalho obtida, há uma boa chance que juntamente com todos os arquivos e diretórios versionados estão outros arquivos e diretórios que não são versionados e nem pretendem ser. Editores de texto deixam diretórios com arquivos de backup. Compiladores de software produzem arquivos intermediários—ou mesmo definitivos—que você normalmente não faria controle de versão. E os próprios usuários deixam vários outros arquivos e diretórios, sempre que acharem adequado, muitas vezes em cópias de trabalho com controle de versão.

É ridículo esperar que cópias de trabalho do Subversion sejam de algum modo impenetráveis a este tipo de resíduo e impureza. De fato, o Subversion os considera como um *recurso* que suas cópias de trabalho estão apenas com diretórios normais, como árvores não-versionadas. Mas estes arquivos e diretórios que não deveriam ser versionados podem causar algum incômodo aos usuários do Subversion. Por exemplo, pelo fato dos comandos **svn add** e **svn import** agirem recursivamente por padrão, e não saberem quais arquivos em uma dada árvore você deseja ou não versionar, é acidentalmente fácil adicionar coisas ao controle de versão que você não pretendia. E pelo fato do comando **svn status** reportar, por padrão, cada item de interesse em uma cópia de trabalho—incluindo arquivos e diretórios não versionados—sua saída pode ficar muito poluída, onde grande número destas coisas aparecem.

Portanto, o Subversion oferece dois meios para dizer quais arquivos você preferiria que ele simplesmente desconsiderasse. Um dos meios envolve o uso do sistema de configuração do ambiente de execução do Subversion (veja “Área de Configuração do Tempo de Execução”), e conseqüentemente aplica-se a todas operações do Subversion que fazem uso desta configuração do ambiente de execução, geralmente aquelas executadas em um computador específico, ou por um

usuário específico de um computador. O outro meio faz uso do suporte de propriedade de diretório do Subversion, é mais fortemente vinculado à própria árvore versionada e, conseqüentemente, afeta todos aqueles que têm uma cópia de trabalho desta árvore. Os dois mecanismos usam filtros de arquivo.

O sistema de configuração *runtime* do Subversion oferece uma opção, `global-ignores`, cujo valor é uma coleção de filtros de arquivo delimitados por espaços em branco (também conhecida com *globs*). O cliente do Subversion verifica esses filtros em comparação com os nomes dos arquivos que são candidatos para adição ao controle de versão, bem como os arquivos não versionados os quais o comando **svn status** notifica. Se algum nome de arquivo coincidir com um dos filtros, basicamente, o Subversion atuará como se o arquivo não existisse. Isto é realmente útil para os tipos de arquivos que você raramente precisará controlar versão, tal como cópias de arquivos feitas por editores como os arquivos `*~` e `.*~` do *Emacs*.

Quando encontrada em um diretório versionado, a propriedade `svn:ignore` espera que contenha uma lista de filtros de arquivo delimitadas por quebras de linha que o Subversion deve usar para determinar objetos ignoráveis neste mesmo diretório. Estes filtros não anulam os encontrados na opção `global-ignores` da configuração *runtime*, porém, são apenas anexados a esta lista. E é importante notar mais uma vez que, ao contrário da opção `global-ignores`, os filtros encontrados na propriedade `svn:ignore` aplicam-se somente ao diretório no qual esta propriedade está definida, e em nenhum de seus subdiretórios. A propriedade `svn:ignore` é uma boa maneira para dizer ao Subversion ignorar arquivos que estão susceptíveis a estarem presentes em todas as cópias de trabalho de usuário deste diretório, assim como as saídas de compilador ou—para usar um exemplo mais apropriado para este livro—os arquivos HTML, PDF, ou PostScript produzidos como o resultado de uma conversão de alguns arquivos XML do fonte DocBook para um formato de saída mais legível.

O suporte do Subversion para filtros de arquivos ignoráveis estende somente até o processo de adicionar arquivos e diretórios não versionados ao controle de versão. Desde que um objeto está sob o controle do Subversion, os mecanismos de filtro de ignoração já não são mais aplicáveis a ele. Em outras palavras, não espere que o Subversion deixe de realizar a submissão de mudanças que você efetuou em arquivos versionados simplesmente porque estes nomes de arquivo coincidem com um filtro de ignoração—o Subversion *sempre* avisa quais objetos foram versionados.

Filtros de Rejeição para Usuários CVS

A propriedade `svn:ignore` do Subversion é muito similar em sintaxe e função ao arquivo `.cvsignore` do CVS. De fato, se você está migrando de uma cópia de trabalho CVS para Subversion, você pode migrar os filtros de rejeição, diretamente, pelo uso do arquivo `.cvsignore` como arquivo de entrada para o comando **svn propset**:

```
$ svn propset svn:ignore -F .cvsignore .
property 'svn:ignore' set on '.'
$
```

Existem, entretanto, algumas diferenças nos meios que CVS e Subversion manipulam filtros de rejeição. Os dois sistemas usam os filtros de rejeição em tempos um pouco diferentes, e existem ligeiras discrepâncias na aplicação dos filtros de rejeição. Além disso, o Subversion não reconhece o uso do filtro `!` como uma redefinição que torna os filtros seguintes como não-ignorados.

A lista global de filtros de rejeição tende ser mais uma questão de gosto pessoal, e vinculada mais estreitamente a uma série de ferramentas específicas do usuário do que aos detalhes de qualquer cópia de trabalho particular necessita. Assim, o resto desta seção focará na propriedade `svn:ignore` e seus usos.

Digamos que você tenha a seguinte saída do **svn status**:

```
$ svn status calc
 M     calc/button.c
?      calc/calculator
?      calc/data.c
?      calc/debug_log
?      calc/debug_log.1
?      calc/debug_log.2.gz
?      calc/debug_log.3.gz
```

Neste exemplo, você realizou algumas modificações no arquivo `button.c`, mas em sua cópia de trabalho você também possui alguns arquivos não-versionados: o mais recente programa `calculator` que você compilou a partir do seu código fonte, um arquivo fonte nomeado `data.c`, e uma série de arquivos de registro da saída de depuração. Agora, você sabe que seu sistema de construção sempre resulta no programa `calculator` como produto. [5] E você sabe que sua ferramenta de testes sempre deixa aqueles arquivos de registro de depuração alojando ao redor. Estes fatos são verdadeiros para todas cópias de trabalho deste projeto, não para apenas sua própria. E você também não está interessado em ver aquelas coisas toda vez que você executa **svn status**, e bastante seguro que ninguém mais está interessado em nenhuma delas. Sendo assim, você executa **svn propedit svn:ignore calc** para adicionar alguns filtros de rejeição para o diretório `calc`. Por exemplo, você pode adicionar os filtros abaixo como o novo valor da propriedade `svn:ignore`:

```
calculator
debug_log*
```

Depois de você adicionar esta propriedade, você terá agora uma modificação de propriedade local no diretório `calc`. Mas note que o restante da saída é diferente para o comando **svn status**:

```
$ svn status
 M     calc
 M     calc/button.c
?      calc/data.c
```

Agora, todas aqueles resíduos não são apresentados nos resultados! Certamente, seu programa compilado `calculator` e todos aqueles arquivos de registro estão ainda em sua cópia de trabalho. O Subversion está simplesmente não lembrando você que eles estão presentes e não-versionados. E agora com todos os arquivos desinteressantes removidos dos resultados, você visualizará somente os itens mais interessantes—assim como o arquivo de código fonte `data.c` que você provavelmente esqueceu de adicionar ao controle de versão.

Evidentemente, este relatório menos prolixo da situação de sua cópia de trabalho não é a única disponível. Se você realmente quiser ver os arquivos ignorados como parte do relatório de situação, você pode passar a opção `--no-ignore` para o Subversion:

```
$ svn status --no-ignore
 M     calc
 M     calc/button.c
I      calc/calculator
?      calc/data.c
I      calc/debug_log
I      calc/debug_log.1
I      calc/debug_log.2.gz
I      calc/debug_log.3.gz
```

Como mencionado anteriormente, a lista de filtros de arquivos a ignorar também é usada pelos comandos **svn add** e **svn import**. Estas duas operações implicam solicitar ao Subversion iniciar o

[5] Não é isso o resultado completo de um sistema de construção?

gerenciamento de algum conjunto de arquivos e diretórios. Ao invés de forçar o usuário a escolher quais arquivos em uma árvore ele deseja iniciar o versionamento, o Subversion usa os filtros de rejeição—tanto a lista global quanto a por diretório (`svn-ignore`)—para determinar quais arquivos não devem ser varridos para o sistema de controle de versão como parte de uma operação recursiva de adição ou importação. E da mesma forma, você pode usar a opção `--no-ignore` para indicar ao Subversion desconsiderar suas listas de rejeição e operar em todos os arquivos e diretórios presentes.

Substituição de Palavra-Chave

O Subversion possui a capacidade de substituir *palavras-chave*—pedaços de informação úteis e dinâmicos sobre um arquivo versionado—dentro do conteúdo do próprio arquivo. As palavras-chave geralmente fornece informação sobre a última modificação realizada no arquivo. Pelo fato desta informação modificar toda vez que o arquivo é modificado, e mais importante, apenas *depois* que o arquivo é modificado, isto é um aborrecimento para qualquer processo a não ser que o sistema de controle de versão mantenha os dados completamente atualizados. Se deixada para os autores humanos, a informação se tornaria inevitavelmente obsoleta.

Por exemplo, digamos que você tem um documento no qual gostaria de mostrar a última data em que ele foi modificado. Você poderia obrigar que cada autor deste documento que, pouco antes de submeter suas alterações, também ajustasse a parte do documento que descreve quando ele fez a última alteração. Porém, mais cedo ou mais tarde, alguém esqueceria de fazer isto. Em vez disso, basta solicitar ao Subversion que efetue a substituição da palavra-chave `LastChangedDate` pelo valor adequado. Você controla onde a palavra-chave é inserida em seu documento colocando uma *âncora de palavra-chave* no local desejado dentro do arquivo. Esta âncora é apenas uma sequência de texto formatada como `$`*`NomeDaPalavraChave`*`$`.

Todas as palavras-chave são sensíveis a minúsculas e maiúsculas onde aparecem como âncoras em arquivos: você deve usar a capitalização correta para que a palavra-chave seja expandida. Você deve considerar que o valor da propriedade `svn:keywords` esteja ciente da capitalização também—certos nomes de palavras-chave serão reconhecidos, independentemente do caso, mas este comportamento está desaprovado.

O Subversion define a lista de palavras-chave disponíveis para substituição. Esta lista contém as seguintes cinco palavras-chave, algumas das quais possuem apelidos que você pode também utilizar:

`Date`
: Esta palavra-chave descreve a última vez conhecida em que o arquivo foi modificado no repositório, e está na forma `$Date: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006) $`. Ela também pode ser especificada como `LastChangedDate`.

`Revision`
: Esta palavra-chave descreve a última revisão conhecida em que este arquivo foi modificado no repositório, e é apresentada na forma `$Revision: 144 $`. Ela também pode ser especificada como `LastChangedRevision` ou `Rev`.

`Author`
: Esta palavra-chave descreve o último usuário conhecido que modificou este arquivo no repositório, e é apresentada na forma `$Author: harry $`. Ela também pode ser especificada como `LastChangedBy`.

`HeadURL`
: Esta palavra-chave descreve a URL completa para a versão mais recente do arquivo no repositório, e é apresentada na forma `$HeadURL: http://svn.collab.net/repos/trunk/README $`. Ela também pode ser abreviada como `URL`.

`Id`
: Esta palavra-chave é uma combinação comprimida das outras palavras-chave. Sua substituição apresenta-se como `$Id: calc.c 148 2006-07-28 21:30:43Z sally $`, e é interpretada

no sentido de que o arquivo `calc.c` foi modificado pela última vez na revisão 148 na noite de 28 de julho de 2006 pelo usuário `sally`.

Muitas das descrições anteriores usam a frase “último valor conhecido” ou algo parecido. Tenha em mente que a expansão da palavra-chave é uma operação no lado do cliente, e seu cliente somente “conhece” sobre mudanças que tenham ocorrido no repositório quando você atualiza sua cópia de trabalho para incluir essas mudanças. Se você nunca atualizar sua cópia de trabalho, suas palavras-chave nunca expandirão para valores diferentes, mesmo que esses arquivos versionados estejam sendo modificados regularmente no repositório.

Simplesmente adicionar texto da âncora de uma palavra-chave em seu arquivo faz nada de especial. O Subversion nunca tentará executar substituições textuais no conteúdo de seu arquivo a não ser que seja explicitamente solicitado. Afinal, você pode estar escrevendo um documento [6] sobre como usar palavras-chave, e você não quer que o Subversion substitua seus belos exemplos de âncoras de palavra-chave, permanecendo não-substituídas!

Para dizer ao Subversion se substitui ou não as palavras-chave em um arquivo particular, voltamos novamente aos subcomandos relacionados a propriedades. A propriedade `svn:keywords`, quando definida em um arquivo versionado, controla quais palavras-chave serão substituídas naquele arquivo. O valor é uma lista delimitada por espaços dos nomes ou apelidos de palavra-chave encontradas na tabela anterior.

Por exemplo, digamos que você tenha um arquivo versionado nomeado `weather.txt` que possui esta aparência:

```
Aqui está o mais recente relatório das linhas iniciais.
$LastChangedDate$
$Rev$
Acúmulos de nuvens estão aparecendo com mais freqüência quando o verão se aproxima.
```

Sem definir a propriedade `svn:keywords` neste arquivo, o Subversion fará nada especial. Agora, vamos permitir a substituição da palavra-chave `LastChangedDate`.

```
$ svn propset svn:keywords "Date Author" weather.txt
property 'svn:keywords' set on 'weather.txt'
$
```

Agora você fez uma modificação local da propriedade no arquivo `weather.txt`. Você verá nenhuma mudança no conteúdo do arquivo (ao menos que você tenha feito alguma definição na propriedade anteriormente). Note que o arquivo continha uma âncora de palavra-chave para a palavra-chave `Rev`, no entanto, não incluímos esta palavra-chave no valor da propriedade que definimos. Felizmente, o Subversion ignorará pedidos para substituir palavras-chave que não estão presentes no arquivo, e não substituirá palavras-chave que não estão presentes no valor da propriedade `svn:keywords`.

Imediatamente depois de você submeter esta mudança de propriedade, o Subversion atualizará seu arquivo de trabalho com o novo texto substituto. Em vez de ver a sua âncora da palavra-chave `$LastChangedDate$`, você verá como resultado seu valor substituído. Este resultado também contém o nome da palavra-chave, que continua sendo limitada pelos caracteres de sinal de moeda ($). E como prevíamos, a palavra-chave `Rev` não foi substituída porque não solicitamos que isto fosse realizado.

Note também, que definimos a propriedade `svn:keywords` para “Date Author” e, no entanto, a âncora da palavra-chave usou o apelido `$LastChangedDate$` e ainda sim expandiu corretamente.

```
Aqui está o mais recente relatório das linhas iniciais.
```

[6] … ou até mesmo uma seção de um livro …

```
$LastChangedDate: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006) $
$Rev$
Acúmulos de nuvens estão aparecendo com mais freqüência quando o verão se aproxima.
```

Se agora alguém submeter uma mudança para `weather.txt`, sua cópia deste arquivo continuará a mostrar o mesmo valor para a palavra-chave substituída como antes—até que você atualize sua cópia de trabalho. Neste momento as palavras-chave em seu arquivo `weather.txt` serão re-substituídas com a informação que reflete a mais recente submissão conhecida para este arquivo.

Onde está $GlobalRev$?

Novos usuários são freqüentemente confundidos pela forma que a palavra-chave `$Rev$` trabalha. Como o repositório possui um número de revisão único, globalmente incrementado, muitas pessoas assumem que este número está refletido no valor da palavra-chave `$Rev$`. Porém, `$Rev$` reflete a última revisão na qual o arquivo foi *modificado*, não a última revisão para qual ele foi atualizado. Compreender isto esclarece a confusão, mas a frustração muitas vezes permanece—sem o suporte de uma palavra-chave do Subversion para isso, como podemos obter automaticamente o número de revisão global em seus arquivos?

Para fazer isto, você precisa de processamento externo. O Subversion vem com uma ferramenta chamada **svnversion** que foi projetada apenas para este propósito. O comando **svnversion** rastreia sua cópia de trabalho e produz como saída as revisões que encontra. Você pode usar este programa, mais algumas outras ferramentas, para embutir esta informação sobre as revisões globais em seus arquivos. Para mais informações sobre **svnversion**, veja “svnversion”.

O Subversion 1.2 introduziu uma nova variante da sintaxe de palavra-chave que trouxe funcionalidade adicional e útil—embora talvez atípica. Agora você pode dizer ao Subversion para manter um tamanho fixo (em termos do número de bytes consumidos) para a palavra-chave substituída. Pelo uso de um duplo dois pontos (`::`) após o nome da palavra-chave, seguido por um número de caracteres de espaço, você define esta largura fixa. Quando o Subversion for substituir sua palavra-chave para a palavra-chave e seu valor, ele substituirá essencialmente apenas aqueles caracteres de espaço, deixando a largura total do campo da palavra-chave inalterada. Se o valor substituído for menor que a largura definida para o campo, haverá caracteres de enchimento extras (espaços) no final do campo substituído; se for mais longo, será truncado com um caractere de contenção especial (#) logo antes do sinal de moeda delimitador de fim.

Por exemplo, digamos que você possui um documento em que temos alguma seção com dados tabulares refletindo as palavras-chave do Subversion sobre o documento. Usando a sintaxe de substituição de palavra-chave original do Subversion, seu arquivo pode parecer com alguma coisa como:

```
$Rev$:      Revisão da última submissão
$Author$:   Autor da última submissão
$Date$:     Data da última submissão
```

Neste momento, vemos tudo de forma agradável e tabular. Mas quando você em seguida submete este arquivo (com a substituição de palavra-chave habilitada, certamente), vemos:

```
$Rev: 12 $:     Revisão da última submissão
$Author: harry $:  Autor da última submissão
$Date: 2006-03-15 02:33:03 -0500 (Wed, 15 Mar 2006) $:    Data da última submissão
```

O resultado não é tão elegante. E você pode ser tentado a então ajustar o arquivo depois da substituição para que pareça tabular novamente. Mas isto apenas funciona quando os valores da palavra-chave são da mesma largura. Se a última revisão submetida aumentar em uma casa decimal (ou seja, de 99

para 100), ou se uma outra pessoa com um nome de usuário maior submete o arquivo, teremos tudo bagunçado novamente. No entanto, se você está usando o Subversion 1.2 ou superior, você pode usar a nova sintaxe para palavra-chave com tamanho fixo, definir algumas larguras de campo que sejam razoáveis, e agora seu arquivo pode ter esta aparência:

```
$Rev::               $:  Revisão da última submissão
$Author::            $:  Autor da última submissão
$Date::              $:  Data da última submissão
```

Você submete esta mudança ao seu arquivo. Desta vez, o Subversion nota a nova sintaxe para palavra-chave com tamanho fixo, e mantém a largura dos campos como definida pelo espaçamento que você colocou entre o duplo dois pontos e o sinal de moeda final. Depois da substituição, a largura dos campos está completamente inalterada—os curtos valores de `Rev` e `Author` são preenchidos com espaços, e o longo campo `Date` é truncado com um caractere de contenção:

```
$Rev:: 13            $:  Revisão da última submissão
$Author:: harry      $:  Autor da última submissão
$Date:: 2006-03-15 0#$:  Data da última submissão
```

O uso de palavras-chave de comprimento fixo é especialmente útil quando executamos substituições em formatos de arquivo complexos que por si mesmo usam campos de comprimento fixo nos dados, ou que o tamanho armazenado de um determinado campo de dados é predominantemente difícil de modificar fora da aplicação original do formato (assim como para documentos do Microsoft Office).

Esteja ciente que pelo fato da largura do campo de uma palavra-chave é medida em bytes, o potencial de corrupção de valores de multi-byte existe. Por exemplo, um nome de usuário que contém alguns caracteres multi-byte em UTF-8 pode sofrer truncamento no meio da seqüência de bytes que compõem um desses caracteres. O resultado será um mero truncamento quando visualizado à nível de byte, mas provavelmente aparecerá como uma cadeia com um caractere adicional incorreto ou ilegível quando exibido como texto em UTF-8. É concebível que certas aplicações, quando solicitadas a carregar o arquivo, notariam o texto em UTF-8 quebrado e ainda considerem todo o arquivo como corrompido, recusando-se a operar sobre o arquivo de um modo geral. Portanto, ao limitar palavras-chave para um tamanho fixo, escolha um tamanho que permita este tipo de expansão ciente dos bytes.

Travamento

O modelo de controle de versão copiar-modificar-fundir do Subversion ganha e perde sua utilidade em seus algoritmos de fusão de dados, especificamente sobre quão bem esses algoritmos executam ao tentar resolver conflitos causados por múltiplos usuários modificando o mesmo arquivo simultaneamente. O próprio Subversion oferece somente um algoritmo, um algoritmo de diferenciação de três meios, que é inteligente o suficiente para manipular dados até uma granularidade de uma única linha de texto. O Subversion também permite que você complemente o processamento de fusão de conteúdo com utilitários de diferenciação externos (como descrito em “Ferramentas diff3 Externas”), alguns dos quais podem fazer um trabalho ainda melhor, talvez oferecendo granularidade em nível de palavra ou em nível de caractere de texto. Mas o comum entre esses algoritmos é que eles geralmente trabalham apenas sobre arquivos de texto. O cenário começa a parecer consideravelmente rígido quando você começa a discursar sobre fusões de conteúdo em formatos de arquivo não-textual. E quando você não pode encontrar uma ferramenta que possa manipular este tipo de fusão, você começa a verificar os problemas com o modelo copiar-modificar-fundir.

Vejamos um exemplo da vida real onde este modelo não trabalha adequadamente. Harry e Sally são ambos desenhistas gráficos trabalhando no mesmo projeto, que faz parte do marketing paralelo para um automóvel mecânico. O núcleo da concepção de um determinado cartaz é uma imagem de um carro que necessita de alguns reparos, armazenada em um arquivo usando o formato de imagem PNG.

O leiaute do cartaz está quase pronto, e tanto Harry quanto Sally estão satisfeitos com a foto que eles escolheram para o carro danificado—um Ford Mustang 1967 azul bebê com uma parte infelizmente amassada no pára-lama dianteiro esquerdo.

Agora, como é comum em trabalhos de desenho gráfico, existe uma mudança de planos que faz da cor do carro uma preocupação. Então, Sally atualiza sua cópia de trabalho para a revisão `HEAD`, inicializa seu software de edição de fotos, e realiza alguns ajustes na imagem de modo que o carro está agora vermelho cereja. Enquanto isso, Harry, sentindo-se particularmente inspirado neste dia, decide que a imagem teria mais impacto se o carro também apresentasse ter sofrido um maior impacto. Ele, também, atualiza para a revisão `HEAD`, e então, desenha algumas rachaduras no pára-brisa do veículo. Ele conduz de forma a concluir seu trabalho antes de Sally terminar o dela, e depois, admirando o fruto de seu inegável talento, submete a imagem modificada. Pouco tempo depois, Sally finaliza sua nova versão do carro, e tenta submeter suas mudanças. Porém, como esperado, o Subversion falha na submissão, informando Sally que agora sua versão da imagem está desatualizada.

Vejamos onde a dificuldade ocorre. Se Harry e Sally estivessem realizando mudanças em um arquivo de texto, Sally iria simplesmente atualizar sua cópia de trabalho, recebendo as mudanças que Harry realizou. No pior caso possível, eles teriam modificado a mesma região do arquivo, e Sally teria que realizar uma adequada resolução do conflito. Mas estes não são arquivos de texto—são imagens binárias. E enquanto seja uma simples questão de descrever o que seria esperado como resultado desta fusão de conteúdos, existe uma pequena chance preciosa de que qualquer software existente seja inteligente o suficiente para examinar a imagem que cada um dos artistas gráficos se basearam para realizarem seu trabalho, as mudanças que Harry fez e as mudanças que Sally faz, e produzir uma imagem de um Mustang vermelho degradado com um pára-brisa trincado!

Obviamente, as coias teriam sido mais simples se Harry e Sally tivessem seqüenciado suas modificações na imagem—se, digamos, Harry aguardasse para desenhar seus trincados no pára-brisa no novo carro vermelho de Sally, ou se Sally trocasse a cor de um carro cujo pára-brisa já estivesse trincado. Como é discutido em “A Solução Copy-Modify-Merge”, a maioria destes tipos de problemas desaparecerão totalmente quando existir uma perfeita comunicação entre Harry e Sally. [7] Porém, como um sistema de controle de versão é de fato uma forma de comunicação, ter um software que facilita a a serialização de esforços não passíveis de paralelamento não é ruim. É neste cenário que a implementação do Subversion do modelo travar-modificar-destravar ganha maior destaque. Este é o momento que falamos sobre a característica de *travamento* do Subversion, a qual é similar aos mecanismos de “obter cópias reservadas” de outros sistemas de controle de versão.

A funcionalidade de travamento do Subversion serve dois propósitos principais:

- *Serializar o acesso a um objeto versionado*. Ao permitir que um usuário requeira programaticamente o direito exclusivo de modificar um arquivo no repositório, este usuário pode estar razoavelmente seguro de que os esforços investidos nas mudanças não-mescláveis não serão desperdiçados—a submissão de suas alterações será bem sucedida.

- *Ajudar a comunicação*. Ao alertar outros usuários que a serialização está em vigor para um determinado objeto versionado, estes outros usuários podem razoavelmente esperar que o objeto está prestes de ser modificado por outra pessoa, e eles, também, podem evitar o desperdício de seu tempo e energia em mudanças não-mescláveis que não serão submetidas adequadamente e ocasionando possível perda de dados.

Quando nos referimos à funcionalidade de travamento do Subversion, estaremos também falando sobre uma coleção de comportamentos bastante diversificada que incluem a capacidade de travar um arquivo [8] versionado (requerendo o direito exclusivo de modificar o arquivo), de destravar este arquivo (cedendo este direito exclusivo de modificar), de ver relatórios sobre quais arquivos estão travados e por quem, de marcar arquivos para os quais o travamento antes da edição é fortemente aconselhado, e assim por diante. Nesta seção, cobriremos todas destas facetas da ampla funcionalidade de travamento.

[7] A comunicação não teria sido algo tão ruim para os homônimos de Harry e Sally em Hollywood, ainda que seja para nosso caso.
[8] Atualmente o Subversion does não permite travas em diretórios.

Os três significados de “trava”

Nesta seção, e em quase todas neste livro, as palavras “trava” e “travamento” representam um mecanismo para exclusão mútua entre os usuários para evitar submissões conflitantes. Infelizmente, existem dois outros tipos de “trava” com os quais o Subversion, e portanto este livro, algumas vezes precisam se preocupar.

O primeiro tipo são as *travas da cópia de trabalho*, usadas internamente pelo Subversion para prevenir conflitos entre múltiplos clientes Subversion operando na mesma cópia de trabalho. Este é o tipo de trava indicada por um `L` na terceira coluna da saída produzida por **svn status**, e removida pelo comando **svn cleanup**, como especificado em “Às Vezes Você Só Precisa Limpar”.

Em segundo lugar, existem as *travas do banco de dados*, usadas internamente pelo sistema Berkeley DB para prevenir conflitos entre múltiplos programas tentando acessar o banco de dados. Este é o tipo de trava cuja indesejável persistência após um erro pode fazer com que um repositório seja “corrompido”, como descrito em “Berkeley DB Recovery”.

Você pode geralmente esquecer destes outros tipos de travas até que algo de errado ocorra e requeira seus cuidados sobre eles. Neste livro, “trava” possui o significado do primeiro tipo ao menos que o contrário esteja claro pelo contexto ou explicitamente indicado.

Criando travas

No repositório Subversion, uma *trava* é um pedaço de metadados que concede acesso exclusivo para um usuário modificar um arquivo. Este usuário é chamado de *proprietário da trava*. Cada trava também tem um identificador único, tipicamente uma longa cadeia de caracteres, conhecida como o *sinal de trava*. O repositório gerencia as travas, basicamente manipulando sua criação, aplicação e remoção. Se qualquer transação de submissão tenta modificar ou excluir um arquivo travado (ou excluir um dos diretórios pais do arquivo), o repositório exigirá dois pedaços de informação—que o cliente executante da submissão esteja autenticado como o proprietário da trava, e que o sinal de trava tenha sido fornecido como parte do processo de submissão como um tipo de prova que o cliente conhece qual trava ele está usando.

Para demonstrar a criação de uma trava, vamos voltar ao nosso exemplo de múltiplos desenhistas gráficos trabalhando sobre os mesmos arquivos binários de imagem. Harry decidiu modificar uma imagem JPEG. Para prevenir que outras pessoas submetessem mudanças no arquivo enquanto ele está modificando-o (bem como alertando-os que ele está prestes a mudá-lo), ele trava o arquivo no repositório usando o comando **svn lock**.

```
$ svn lock banana.jpg -m "Editando arquivo para a liberação de amanhã."
'banana.jpg' locked by user 'harry'.
$
```

Existe uma série de novas coisas demonstradas no exemplo anterior. Primeiro, note que Harry passou a opção `--message (-m)` para o comando **svn lock**. Similar ao **svn commit**, o comando **svn lock** pode receber comentários (seja via `--message (-m)` ou `--file (-F)`) para descrever a razão do travamento do arquivo. Ao contrário do **svn commit**, entretanto, o **svn lock** não exigirá uma mensagem executando seu editor de texto preferido. Os comentários de trava são opcionais, mas ainda recomendados para ajudar na comunicação.

Em segundo lugar, a trava foi bem sucedida. Isto significa que o arquivo não estava travado, e que Harry tinha a mais recente versão do arquivo. Se o arquivo da cópia de trabalho de Harry estivesse desatualizado, o repositório teria rejeitado a requisição, forçando Harry a executar **svn update** e tentar o comando de travamento novamente. O comando de travamento também teria falhado se o arquivo já estivesse travado por outro usuário.

Como você pode ver, o comando **svn lock** imprime a confirmação do sucesso no travamento. A partir deste ponto, o fato de que o arquivo está travado torna-se aparente na saída dos relatórios dos subcomandos **svn status** e **svn info**.

```
$ svn status
     K banana.jpg

$ svn info banana.jpg
Path: banana.jpg
Name: banana.jpg
URL: http://svn.example.com/repos/project/banana.jpg
Repository UUID: edb2f264-5ef2-0310-a47a-87b0ce17a8ec
Revision: 2198
Node Kind: file
Schedule: normal
Last Changed Author: frank
Last Changed Rev: 1950
Last Changed Date: 2006-03-15 12:43:04 -0600 (Wed, 15 Mar 2006)
Text Last Updated: 2006-06-08 19:23:07 -0500 (Thu, 08 Jun 2006)
Properties Last Updated: 2006-06-08 19:23:07 -0500 (Thu, 08 Jun 2006)
Checksum: 3b110d3b10638f5d1f4fe0f436a5a2a5
Lock Token: opaquelocktoken:0c0f600b-88f9-0310-9e48-355b44d4a58e
Lock Owner: harry
Lock Created: 2006-06-14 17:20:31 -0500 (Wed, 14 Jun 2006)
Lock Comment (1 line):
Editando arquivo para a liberação de amanhã.

$
```

O comando **svn info**, o qual não consulta o repositório quando executa sobre caminhos de uma cópia de trabalho, pode mostrar o sinal de trava e revela um importante fato sobre o sinal de trava—que eles são colocados em cache na cópia de trabalho. A presença do sinal de trava é crítica. Ele dá à cópia de trabalho a autorização para fazer uso da trava mais tarde. Além disso, o comando **svn status** mostra um `K` próximo ao arquivo (abreviação para *locKed*), indicando que o sinal de trava está presente.

Em relação aos sinais de trava

Um sinal de trava não é um sinal de autenticação, tanto como um sinal de *autorização*. O sinal não é um segredo protegido. De fato, um sinal de trava exclusivo é descoberto por qualquer pessoa que execute **svn info URL**. Um sinal de trava é especial somente quando reside dentro de uma cópia de trabalho. Ele é prova de que a trava foi criada em uma cópia de trabalho específica, e não noutra qualquer por algum outro cliente. Apenas se autenticando como o proprietário da trava não é suficiente para prevenir acidentes.

Por exemplo, suponha que você travou um arquivo usando um computador em seu escritório, mas deixou o trabalho antes de concluir suas modificações para esse arquivo. Não deveria ser possível acidentalmente submeter mudanças para esse mesmo arquivo do seu computador de casa mais tarde da noite, simplesmente porque você está autenticado como o proprietário da trava. Em outras palavras, o sinal de trava previne uma parte do software relacionado ao Subversion de invadir o trabalho do outro. (Em nosso exemplo, se você realmente precisa modificar o arquivo de uma cópia de trabalho alternativa, você precisaria *parar* a trava e retravar o arquivo.)

Agora que Harry tem o arquivo `banana.jpg` travado, Sally não poderá modificar ou excluir esse arquivo:

```
$ svn delete banana.jpg
D         banana.jpg
$ svn commit -m "Excluir arquivo sem uso."
Deleting       banana.jpg
svn: Commit failed (details follow):
```

```
svn: DELETE of
'/repos/project/!svn/wrk/64bad3a9-96f9-0310-818a-df4224ddc35d/banana.jpg':
423 Locked (http://svn.example.com)
$
```

Porém Harry, após retocar a tonalidade amarela da banana, é capaz de submeter suas mudanças no arquivo. Isso porque ele se autenticou como o proprietário da trava, e também porque sua cópia de trabalho possui o sinal de trava correto:

```
$ svn status
M    K banana.jpg
$ svn commit -m "Torna a banana mais amarela"
Sending        banana.jpg
Transmitting file data .
Committed revision 2201.
$ svn status
$
```

Note que após a submissão ser concluída, **svn status** mostra que o sinal de trava não está mais presente na cópia de trabalho. Este é o comportamento padrão de **svn commit**—ele procura na cópia de trabalho (ou lista de alvos, se você fornecer uma lista desse tipo) por modificações locais, e envia todos os sinalizadores de trava encontrados durante esta caminhada para o servidor como parte da transação de submissão. Após a submissão concluir com sucesso, todas as travas do repositório que forem mencionadas são liberadas—*até mesmo em arquivos que não foram submetidos*. Isto é utilizado para que os usuários não sejam desleixados com os travamentos, ou segurem travas por muito tempo. Se Harry trava de forma desorganizada trinta arquivos em um diretório nomeado `images` porque não tem certeza de quais arquivos ele precisa modificar, por ora apenas modifica quatro destes arquivos, quando ele executar **svn commit images**, o processo mesmo assim liberará todas as trinta travas.

Este comportamento de liberar as travas automaticamente pode ser evitado com a passagem da opção `--no-unlock` ao comando **svn commit**. Isso tem melhor uso para aqueles casos quando você quer submeter mudanças, mas ainda planeja fazer mais mudanças e, portanto, precisa conservar as travas existentes. Você também pode fazer este seu comportamento padrão configurando a opção `no-unlock` do seu ambiente de execução (veja “Área de Configuração do Tempo de Execução”).

Evidentemente, travar um arquivo não o obriga a submeter uma mudança para ele. A trava pode ser liberada a qualquer tempo com um simples comando **svn unlock**:

```
$ svn unlock banana.c
'banana.c' unlocked.
```

Descobrindo as travas

Quando uma submissão falha devido a um trava que outra pessoa criou, é bastante fácil ver os detalhes sobre ela. A forma mais fácil delas é **svn status --show-updates**:

```
$ svn status -u
M              23   bar.c
M    O         32   raisin.jpg
       *       72   foo.h
Status against revision:     105
$
```

Neste exemplo, Sally pode ver não somente que sua cópia de `foo.h` está desatualizada, mas que um dos dois arquivos modificados que ela planeja submeter está travado no repositório. O símbolo

O corresponde a “Other”, significando que existe uma trava sobre o arquivo, e foi criada por outra pessoa. Se ela vier a tentar uma submissão, a trava sobre `raisin.jpg` a impediria. Sally deve estar imaginando quem fez a trava, quando, e porquê. Mais uma vez, **svn info** tem as respostas:

```
$ svn info http://svn.example.com/repos/project/raisin.jpg
Path: raisin.jpg
Name: raisin.jpg
URL: http://svn.example.com/repos/project/raisin.jpg
Repository UUID: edb2f264-5ef2-0310-a47a-87b0ce17a8ec
Revision: 105
Node Kind: file
Last Changed Author: sally
Last Changed Rev: 32
Last Changed Date: 2006-01-25 12:43:04 -0600 (Sun, 25 Jan 2006)
Lock Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Lock Owner: harry
Lock Created: 2006-02-16 13:29:18 -0500 (Thu, 16 Feb 2006)
Lock Comment (1 line):
Necessidade de fazer um ajuste rápido nesta imagem.
$
```

Assim como **svn info** pode ser usado para examinar objetos na cópia de trabalho, ele também pode ser usado para examinar objetos no repositório. Se o argumento principal para **svn info** é um caminho de uma cópia de trabalho, então todas informações em cache da cópia de trabalho são exibidas; qualquer menção a uma trava significa que a cópia de trabalho está mantendo um sinal de trava (se um arquivo é travado por outro usuário ou em outra cópia de trabalho, **svn info** em um caminho de cópia de trabalho não mostrará qualquer informação da trava). Se o argumento principal para **svn info** é uma URL, então as informações refletem a mais recente versão de um objeto no repositório, e qualquer menção a uma trava descreve a atual trava sobre o objeto.

Portanto, neste exemplo particular, Sally pode ver que Harry travou o arquivo em 16 de Fevereiro para “fazer um ajuste rápido”. Já estando em Junho, ela suspeita que ele provavelmente se esqueceu totalmente da trava. Ela poderia ligar para Harry para reclamar e lhe pedir que libere a trava. Se ele estiver indisponível, ela poderá tentar quebrar a trava a força ou solicitar um administrador para o fazer.

Quebrando e roubando travas

Uma trava no repositório não é algo sagrado—na configuração padrão do Subversion, as travas podem ser liberadas não somente pela pessoa que a criou, mas por qualquer outra também. Quando alguém que são seja o criador original da trava a destrói, referimos a isto como *quebrar* a trava.

Para o administrador é simples quebrar travas. Os programas **svnlook** e **svnadmin** possuem a habilidade de mostrar e remover travas diretamente do repositório. (Para mais informações sobre estas ferramentas, veja “An Administrator's Toolkit”.)

```
$ svnadmin lslocks /usr/local/svn/repos
Path: /project2/images/banana.jpg
UUID Token: opaquelocktoken:c32b4d88-e8fb-2310-abb3-153ff1236923
Owner: frank
Created: 2006-06-15 13:29:18 -0500 (Thu, 15 Jun 2006)
Expires:
Comment (1 line):
Ainda melhorando a cor amarela.

Path: /project/raisin.jpg
UUID Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
```

```
Owner: harry
Created: 2006-02-16 13:29:18 -0500 (Thu, 16 Feb 2006)
Expires:
Comment (1 line):
Necessidade de fazer um ajuste rápido nesta imagem.

$ svnadmin rmlocks /usr/local/svn/repos /project/raisin.jpg
Removed lock on '/project/raisin.jpg'.
$
```

Uma opção mais interessante é permitir que usuários quebrem as travas de outros através da rede. Para fazer isto, Sally simplesmente precisa passar a opção `--force` para o comando de destravamento:

```
$ svn status -u
M              23   bar.c
M    O         32   raisin.jpg
       *       72   foo.h
Status against revision:     105
$ svn unlock raisin.jpg
svn: 'raisin.jpg' is not locked in this working copy
$ svn info raisin.jpg | grep URL
URL: http://svn.example.com/repos/project/raisin.jpg
$ svn unlock http://svn.example.com/repos/project/raisin.jpg
svn: Unlock request failed: 403 Forbidden (http://svn.example.com)
$ svn unlock --force http://svn.example.com/repos/project/raisin.jpg
'raisin.jpg' unlocked.
$
```

Agora, a tentativa inicial de Sally para destravar falhou porque ela executou **svn unlock** diretamente em sua cópia de trabalho do arquivo, e nenhum sinal de trava estava presente. Para remover a trava diretamente do repositório, ela precisa passar uma URL para **svn unlock**. Sua primeira tentativa para destravar a URL falhou, porque ela não pode autenticar como a proprietária da trava (nem ela possui o sinal de trava). Mas quando ela passa `--force`, os requisitos de autenticação e autorização são ignorados, e a trava remota agora está quebrada.

Simplesmente quebrar uma trava pode não ser suficiente. No exemplo atual, Sally pode não somente querer quebrar a trava esquecida a longo prazo por Harry, mas também retravar o arquivo para seu próprio uso. Ela pode realizar isto executando **svn unlock --force** e então **svn lock** logo em seguida, mas existe uma pequena chance de que outra pessoa possa travar o arquivo entre os dois comandos. Uma coisa mais simples é *roubar* a trava, que envolve quebrar e retravar o arquivo em um passo atômico. Para fazer isto, Sally passa a opção `--force` para **svn lock**:

```
$ svn lock raisin.jpg
svn: Lock request failed: 423 Locked (http://svn.example.com)
$ svn lock --force raisin.jpg
'raisin.jpg' locked by user 'sally'.
$
```

Em qualquer caso, se a trava é quebrada ou roubada, Harry terá uma surpresa. A cópia de trabalho de Harry ainda contém o sinal original da trava, mas esta trava não existe mais. O sinal da trava está agora *extinto*. A trava representada pelo sinal de trava, ou terá sido quebrada (não está mais no repositório), ou roubada (substituída por uma trava diferente). De qualquer forma, Harry pode ver isto pedindo para **svn status** verificar o repositório:

```
$ svn status
     K raisin.jpg
```

```
$ svn status -u
     B         32   raisin.jpg
$ svn update
  B  raisin.jpg
$ svn status
$
```

Se a trava foi quebrada no repositório, então **svn status --show-updates** exibe um símbolo `B` (*Broken*) próximo ao arquivo. Se uma nova trava existe no lugar da anterior, então um símbolo `T` (*sTolen*) é mostrado. Finalmente, **svn update** relata os sinais de trava existentes e os remove da cópia de trabalho.

Políticas de Travamento

Diferentes sistemas possuem diferentes noções de como rigorosa uma trava deve ser. Algumas pessoas afirmam que travas devem ser estritamente aplicadas a todo custo, liberáveis somente pelo criador original ou administrador. Eles argumentam que se qualquer um pode quebrar uma trava, então o caos corre galopante e toda circunstância de travamento é derrotada. O outro lado afirma que travas são, antes de mais nada, uma ferramenta de comunicação. Se usuários estão constantemente quebrando as travas de outros, então ele representa um fracasso cultural dentro da equipe e o problema sai fora do escopo da aplicação de software.

Por padrão o Subversion possui uma abordagem “branda”, mas ainda permite que administradores criem políticas de aplicação mais rigorosas através da utilização de scripts de gancho. Em particular, os ganchos `pre-lock` e `pre-unlock` permitem aos administradores decidir quando a criação e liberação de travas são autorizadas a acontecer. Dependendo se uma trava já existe ou não, estes dois ganchos podem decidir se permitem ou não que um certo usuário pode quebrar ou roubar uma trava. Os ganchos `post-lock` e `post-unlock` também estão disponíveis, e podem ser usados para enviar e-mail após ações de travamento. Para aprender mais sobre ganhos de repositório, veja “Implementing Repository Hooks”.

Comunicação de Travas

Vimos como **svn lock** e **svn unlock** podem ser usados para criar, liberar, quebrar, e roubar travas. Isso satisfaz o objetivo de serializar o acesso a submissões de um arquivo. Mas o que aconteceu com o maior problema da prevenção de perda de tempo?

Por exemplo, suponha que Harry trave um arquivo de imagem e, em seguida, inicie sua edição. Entretanto, a milhas de distância, Sally deseja fazer a mesma coisa. Ela não pensa em executar **svn status --show-updates**, portanto ele não tem idéia de que Harry já tenha travado o arquivo. Ela gasta horas editando o arquivo, e quando ela tenta submeter sua mudança, ela descobre que ou o arquivo está travado ou que ela está desatualizada. Indiferente disso, suas alterações não são mescláveis com as de Harry. Uma destas duas pessoas tem que jogar fora seu trabalho, e um monte de tempo foi perdido.

A solução do Subversion para este problema é oferecer um mecanismo para avisar aos usuários que um arquivo deve ser travado *antes* de iniciar sua edição. O mecanismo é uma propriedade especial, `svn:needs-lock`. Se esta propriedade está anexada a um arquivo (indiferente de seu valor, o qual é irrelevante), então o Subversion tentará utilizar permissões a nível de sistema de arquivo para tornar o arquivo somente leitura—exceto, claro, o usuário tiver explicitamente travado o arquivo. Quando um sinal de trava está presente (como resultado de executar **svn lock**), o arquivo fica como leitura e escrita. Quando a trava é liberada, o arquivo fica como somente leitura novamente.

A teoria, então, é que se o arquivo de imagem tem esta propriedade anexada, então Sally iria verificar imediatamente que alguma coisa está estranho quando ela abrir o arquivo para edição: muitas aplicações avisam os usuários imediatamente quando um arquivo somente leitura é aberto para edição, e quase todos impedem que alterações sejam salvas no arquivo. Isto lembra ela para travar o arquivo antes de editá-lo, e então ela descobre a trava já existente:

```
$ /usr/local/bin/gimp raisin.jpg
gimp: error: file is read-only!
$ ls -l raisin.jpg
-r--r--r--   1 sally   sally   215589 Jun  8 19:23 raisin.jpg
$ svn lock raisin.jpg
svn: Lock request failed: 423 Locked (http://svn.example.com)
$ svn info http://svn.example.com/repos/project/raisin.jpg | grep Lock
Lock Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Lock Owner: harry
Lock Created: 2006-06-08 07:29:18 -0500 (Thu, 08 June 2006)
Lock Comment (1 line):
Fazendo alguns ajustes.  Travando para as próximas duas horas.
$
```

Tanto usuários e administradores são encorajados a anexar a propriedade `svn:needs-lock` em qualquer arquivo que não possa ser contextualmente mesclado. Esta é a principal técnica para incentivar bons hábitos de travamento e evitar desperdício de esforços.

Note que esta propriedade é um instrumento de comunicação que trabalha independentemente do sistema de travamento. Em outras palavras, qualquer arquivo pode ser travado, estando esta propriedade presente ou não. E reciprocamente, a presença desta propriedade não faz com que o repositório requeira uma trava quando for submeter as mudanças.

Infelizmente, o sistema não é perfeito. É possível que mesmo quando um arquivo possua a propriedade, a advertência de somente leitura nem sempre funcione. Algumas vezes as aplicações comportam-se mal e “adulteram” o arquivo somente leitura, silenciosamente permitindo aos usuários editar e salvar o arquivo de qualquer forma. Não há muito que o Subversion possa fazer nesta situação—de qualquer maneira, simplesmente não há substituição para uma boa comunicação entre as pessoas. [9]

Definições Externas

Às vezes, é útil construir uma cópia de trabalho que é composta por diferentes *checkouts*. Por exemplo, talvez você queira que diferentes subdiretórios venham de diferentes locais em um repositório, ou até mesmo de diferentes repositórios. Você poderia configurar tal cenário manualmente— usando **svn checkout** para criar o tipo de estrutura aninhada de cópia de trabalho que você está tentando construir. Mas, se essa estrutura é importante para todos os que usam seu repositório, todos os outros usuários precisarão realizar as mesmas operações de *checkout* que você fez.

Felizmente, o Subversion provê suporte para *definições externas*. Uma definição externa é um mapeamento de um diretório local para a URL—e, idealmente, uma determinada revisão—de um diretório sob controle de versão. No Subversion, você declara definições externas em conjunto usando a propriedade `svn:externals`. Você pode criar ou modificar essa propriedade usando **svn propset** ou **svn propedit** (veja “Manipulando Propriedades”). Essa propriedade pode ser configurada em qualquer diretório sob controle de versão, e seu valor é uma tabela multilinha de subdiretórios (relativos ao diretório sob controle de versão no qual a propriedade está configurada), opções de revisão, e URLs absolutas (totalmente qualificadas) de repositórios Subversion.

```
$ svn propget svn:externals calc
third-party/sounds             http://sounds.red-bean.com/repos
third-party/skins              http://skins.red-bean.com/repositories/skinproj
third-party/skins/toolkit -r21 http://svn.red-bean.com/repos/skin-maker
```

A conveniência da propriedade `svn:externals` é que, uma vez configurada em um diretório sob controle de versão, qualquer pessoa que obtém uma cópia de trabalho desse diretório também é

[9] Exceto, talvez, uma mente-lógica do clássico Vulcaniano.

beneficiada pelas definições externas. Em outras palavras, uma vez que alguém investiu tempo e esforço para definir essa cópia de trabalho feita de *checkouts* aninhados, ninguém mais precisa se incomodar—o Subversion, através do *checkout* da cópia de trabalho original, também obterá as cópias de trabalho externas.

Os subdiretórios alvos relativos das definições externas *não podem* existir no seu sistema de arquivos nem no de outros usuários—o Subversion irá criá-los quando obter a cópia de trabalho externa.

Note o exemplo anterior de definições externas. Quando alguém obtém uma cópia de trabalho do diretório `calc`, o Subversion também obtém os itens encontrados nas suas definições externas.

```
$ svn checkout http://svn.example.com/repos/calc
A  calc
A  calc/Makefile
A  calc/integer.c
A  calc/button.c
Checked out revision 148.

Fetching external item into calc/third-party/sounds
A  calc/third-party/sounds/ding.ogg
A  calc/third-party/sounds/dong.ogg
A  calc/third-party/sounds/clang.ogg
…
A  calc/third-party/sounds/bang.ogg
A  calc/third-party/sounds/twang.ogg
Checked out revision 14.

Fetching external item into calc/third-party/skins
…
```

Se você precisar mudar as definições externas, você pode fazer isso usando os subcomandos para modificação de propriedades normalmente. Quando você submeter uma alteração na propriedade `svn:externals`, o Subversion irá sincronizar os itens submetidos com as definições externas na próxima vez que você executar um **svn update**. A mesma coisa irá acontecer quando outros atualizarem suas cópias de trabalho e recebam as suas modificações nas definições externas.

Como o valor da propriedade `svn:externals` é um conteúdo de múltiplas linhas, nós recomendamos fortemente que você use o **svn propedit** ao invés do **svn propset**.

Você deveria considerar seriamente o uso de um número de revisão explícito em todas as suas definições externas. Fazer isso significa que você tem que decidir quando trazer um diferente registro instantâneo de uma informação externa, e exatamente qual instantâneo trazer. Além de evitar a surpresa de obter mudanças de repositórios de terceiros sobre as quais você pode não ter nenhum controle, usar número de revisão explícitos também significa que conforme você voltar no tempo sua cópia de trabalho para uma revisão anterior, suas definições externas também serão revertidas para a forma como estavam na revisão passada, o que por sua vez significa que as cópias de trabalho serão atualizadas de volta para corresponder à forma como *elas* se pareciam quando seu repositório estava naquela revisão anterior. Para projetos de software, isso poderia ser a diferença entre uma compilação de sucesso ou uma falha em um momento passado de sua complexa base de código.

O comando **svn status** também reconhece definições externas, exibindo um código de status X para subdiretórios desmembrados nos quais as externas foram obtidas, e então varrer recursivamente dentro destes subdiretórios para mostrar o status dos próprios itens externos.

O suporte que existe para definições externas no Subversion ainda está abaixo do ideal. Primeiro, uma definição externa pode apenas apontar para diretórios, não para arquivos. Segundo, as definições externas não podem apontar para caminhos relativos (tais como `../../skins/myskin`). Terceiro, o suporte a cópias de trabalho criadas por meio de definições externas ainda está desconectado da cópia de trabalho primária (na qual a propriedade `svn:externals` dos diretórios versionados foi atualmente definida). E o Subversion ainda só pode operar verdadeiramente em cópias de trabalho não-desmembradas. Então, por exemplo, se você quiser submeter as alterações que você tenha feito em um ou mais destas cópias de trabalho externas, você deve executar um **svn commit** explicitamente nessas cópias de trabalho—submeter alterações numa cópia de trabalho não irá implicar numa recursão dentro de nenhuma das externas.

E também, como as definições externas em si usam URLs absolutas, a movimentação ou cópia de um diretório ao qual elas estejam anexadas não afetará aquela obtida como uma externa (ainda que o subdiretório local de destino relativo seja, é claro, movido com o diretório renomeado). Isto pode ser confuso—ou mesmo frustrante—em certas situações. Por exemplo, digamos que você tenha um diretório de alto nível chamado `my-project`, e que você tenha criado uma definição externa em um de seus subdiretórios (`my-project/some-dir`) que acompanha a última revisão de outro de seus subdiretórios (`my-project/external-dir`).

```
$ svn checkout http://svn.example.com/projects .
A    my-project
A    my-project/some-dir
A    my-project/external-dir
…
Fetching external item into 'my-project/some-dir/subdir'
Checked out external at revision 11.

Checked out revision 11.
$ svn propget svn:externals my-project/some-dir
subdir http://svn.example.com/projects/my-project/external-dir

$
```

Agora você executa **svn move** para renomear o diretório `my-project` directory. Neste ponto, sua definição externa ainda vai se referir ao caminho relacionado ao diretório `my-project`, muito embora esse diretório não exista mais.

```
$ svn move -q my-project renamed-project
$ svn commit -m "Rename my-project to renamed-project."
Deleting       my-project
Adding         my-renamed-project

Committed revision 12.
$ svn update

Fetching external item into 'renamed-project/some-dir/subdir'
svn: Target path does not exist
$
```

E também, as URLs absolutas que as definições externas usam podem causar problemas com repositórios que estejam disponíveis a partir de múltiplos esquemas de URL. Por exemplo, se seu servidor Subversion estiver configurado para permitir que qualquer um obtenha o conteúdo do repositório por meio de `http://` ou `https://`, mas que apenas possam submeter alterações através de `https://`, você tem um interessante problema em mãos. Se suas definições externas usam o formato `http://` para URLs do repositório, você não será capaz de submeter nada a partir das cópias de trabalho criadas por definições externas. Por outro lado, se elas usam o formato `https://` para URLs, qualquer pessoa que tenha obtido a cópia através de `http://` por não ter um cliente com

suporte a `https://` estará impossibilitado de buscar itens externos. Atente também que se você precisar realocar sua cópia de trabalho (usando **svn switch --relocate**), suas definições externas *não* serão realocadas.

Finalmente, pode ser que algumas vezes você prefira que os subcomandos não reconheçam **svn**, ou mesmo não operem sobre cópias de trabalhos externas. Nesses casos, você pode passar a opção `--ignore-externals` para o subcomando.

Revisões Marcadoras e Revisões Operativas

Nós copiamos, movemos, renomeamos, e substituímos completamente arquivos e diretórios em nossos computadores a todo tempo. E seu sistema de controle de versão não deveria basear em seu modo e fazer estas coisas com seus arquivos e diretórios com versões controladas. O suporte a gerenciamento de arquivos do Subversion é bastante aberto, proporcionando quase tanta flexibilidade para arquivos versionados quanto você desejaria ao manipular os seus não-versionados. Mas essa flexibilidade significa que durante o tempo de vida de seu repositório, um dado objeto versionado pode ter muitos caminhos, e um dado caminho pode representar vários objetos versionados inteiramente diferentes. E isto introduz um certo nível de complexidade em suas interações com esses caminhos e objetos.

O Subversion é muito esperto ao perceber quando uma versão do histórico do objeto inclui tais “mudanças de endereço”. Por exemplo, se você pedir pelo registro do histórico de revisão de um arquivo específico que foi renomeado na última semana, o Subversion felizmente oferece todos estes registros —a revisão na qual a renomeação aconteceu, mais os registros de revisões relevantes tanto antes como depois que foi renomeado. Assim, a maioria das vezes, você não terá que pensar sobre estas coisas. Mas ocasionalmente, o Subversion precisará de sua ajuda para esclarecer ambigüidades.

O exemplo mais simples disto ocorre quando um diretório ou arquivo é excluído do controle de versão, e então um novo diretório ou arquivo é criado com o mesmo nome e adicionado ao controle de versão. Obviamente o que você exclui e o que você depois adicionou não são a mesma coisa. Estas coisas meramente possuíram o mesmo caminho, `/trunk/object` por exemplo. Então, o que significa solicitar ao Subversion o histórico de `/trunk/object`? Você está pedindo sobre o objeto atualmente neste local, ou o antigo objeto que você excluiu deste local? Você está pedindo sobre as operações que aconteceram em *todos* os objetos que alguma vez existiu neste caminho? Obviamente, o Subversion precisa de uma dica do que você realmente quer.

Devido a mudanças regulares, o histórico de um objeto versionado pode ser mais misturado do que isto. Por exemplo, você pode ter um diretório nomeado `concept`, contendo algum projeto de software pessoal em que você esteja brincando. Eventualmente, porém, este projeto amadurece a tal ponto que a idéia parece realmente poder decolar, assim você faz o impensável e decide dar um nome ao projeto. [10] Vamos dizer que você chamou seu software de Frabnaggilywort. Neste ponto, faz sentido renomear o diretório para refletir o novo nome do projeto, assim `concept` é renomeado para `frabnaggilywort`. A vida continua, Frabnaggilywort lança uma versão 1.0, e está sendo baixado e usado diariamente por uma multidão de pessoas que pretendem melhorar suas vidas.

É uma bela história, realmente, mas não termina aqui. Como empreendedor que você é, você já está com novas idéias em mente. Então você cria um novo diretório, `concept`, e o ciclo começa outra vez. De fato, o ciclo recomeça muitas vezes ao longo dos anos, cada vez começando com o antigo diretório `concept`, então algumas vezes vendo esse diretório ser renomeado como você bem o quiser, algumas vezes vendo esse diretório ser excluído quando você descarta a idéia. Ou, para complicar de vez, algumas vezes talvez você `concept` para qualquer outra coisa por algum tempo, mas depois renomei-o de volta para `concept` por alguma razão.

Em cenários como este, tentar instruir o Subversion para trabalhar com estes caminhos reutilizados pode ser um pouco como instruir um motorista dos subúrbios da Chicago ocidental a dirigir sempre a leste na estrada Roosevelt Road e então virar à esquerda na Main Street. Em meros vinte minutos, você pode cruzar com a tal “Main Street” ao andar pela Wheaton, Glen Ellyn ou Lombard. E não, elas

[10]“Você não pretendia dar um nome a ele. Depois que você dá um nome, você começa a ficar ligado a ele.”—Mike Wazowski

não são a mesma rua. Nosso motorista—e o nosso Subversion—precisa de um pouco mais de detalhes para poder fazer a coisa certa.

Na versão 1.1, o Subversion introduziu uma maneira para você dizer exatamente à que Main Street você se refere. É chamada de *revisão marcadora*, e é uma revisão disponibilizada pelo Subversion apenas com propósito de identificar uma linha única de histórico. Como no máximo um objeto versionado pode ocupar um caminho em um dado instante—ou, mais precisamente, em uma dada revisão—a combinação de um caminho e uma revisão marcadora é tudo o que é necessário para se referenciar a uma linha específica de histórico. Revisões marcadoras são especificadas pelo cliente de linha de comando do Subversion usando *sintaxe de arroba*[11], assim chamada porque envolve anexar-se um “sinal de arroba” (`@`) e a revisão marcadora ao final do caminho com o qual a revisão está associada.

Mas e sobre as revisões dadas por `--revision (-r)`, as quais falamos tanto neste livro? Essas revisões (ou conjuntos de revisões) são chamadas de *revisões operativas* (ou *intervalos de revisões operativas*). Uma vez que uma linha em particular do histórico tenha sido identificada usando-se um caminho e uma revisão marcadora, o Subversion executa a operação requisitada usando a(s) revisão(ões) operativa(s). Para relacionar isto com nossa analogia às ruas de Chicago, se nos disserem para irmos para até a Main Street em Wheaton 606 N., [12] poderíamos pensar na “Main Street” como nosso caminho e em “Wheaton” como nossa revisão marcadora. Estes dois pedaços de informação identificam um único caminho que pode ser percorrido (em sentido sul ou sentido norte na Main Street), e que nos permitir andar para cima e para baixo na Main Street ao acaso na busca pelo nosso destino. Agora temos “606 N.” como nossa revisão operativa, de sorte que sabemos *exatamente* aonde temos que ir.

[11]N.T.: Em inglês, o símbolo de arroba é lido como “at”, que tem o sentido de *em* ou *naquele lugar*.
[12]Main Street, Wheaton, 606 N., Illinois, é o endereço do Wheaton History Center. Sacou—“History Center”? Parece apropriado….

O algoritmo de revisões marcadoras

O Subversion em linha de comando executa o algoritmo de revisões marcadora a qualquer momento em que precise resolver possíveis ambigüidades nos caminhos e revisões por ele providos. Aqui está um exemplo de execução:

```
$ svn command -r OPERATIVE-REV item@PEG-REV
```

Se *OPERATIVE-REV* for mais antiga que *PEG-REV*, então o algoritmo será o seguinte:

- Localize o *item* na revisão identificada por *PEG-REV*. Deve ser encontrado apenas um único objeto.
- Trace o histórico pregresso do objeto (através de eventuais renomeações ocorridas) até seu ancestral na revisão *OPERATIVE-REV*.
- Execute a ação requisitada naquele ancestral, onde quer que ele se encontre, ou qualquer que seja o nome que ele tenha ou que tenha tido ao longo do tempo.

Mas e se *OPERATIVE-REV* for *mais recente* que *PEG-REV*? Bem, isso adiciona alguma complexidade ao problema teórico de localização do caminho em *OPERATIVE-REV*, pois o histórico do caminho pode ter sido ramificado várias vezes (graças a operações de cópia) entre *PEG-REV* e *OPERATIVE-REV*. E isso não é tudo—de qualquer maneira, o Subversion não armazena informação o suficiente para traçar eficientemente o histórico das revisões à frente para um objeto. Assim, o algoritmo neste caso é um pouco diferente:

- Localize o *item* na revisão identificada por *OPERATIVE-REV*. Deve ser encontrado apenas um único objeto.
- Trace o histórico pregresso do objeto (através de eventuais renomeações ocorridas) até seu ancestral na revisão *PEG-REV*.
- Verifique se a localização do objeto (caminho) em *PEG-REV* é a mesma que o era na revisão *OPERATIVE-REV*. Se for este o caso, então sabe-se que pelo menos dois locais estão diretamente relacionados, e então execute a ação requisitada na localização em *OPERATIVE-REV*. Caso contrário, nenhuma relação pôde ser estabelecida, então exiba uma mensagem de erro detalhando que nenhuma localização viável foi encontrada. (Algum dia esperamos que o Subversion será capaz de lidar com este cenário de uso com mais graça e flexibilidade.)

Note que mesmo quando você não informa uma revisão marcadora ou uma revisão operativa, elas ainda estarão presentes. Para sua conveniência, `BASE` é tida como revisão marcadora padrão para itens em sua cópia de trabalho e `HEAD` o é para URLs do repositório. E quando nenhuma revisão operativa for informada, por padrão será usada a mesma que a da revisão marcadora.

Digamos que tenhamos criado nosso repositório muito tempo atrás, e que na revisão 1 adicionamos nosso primeiro diretório `concept`, além de um arquivo `IDEA` nesse diretório contendo as idéias relacionadas ao conceito. Depois de algumas revisões nas quais códigos reais foram adicionados e manipulados, nós, na revisão 20, renomeamos este diretório para `frabnaggilywort`. Lá pela revisão 27, temos um novo conceito, e criamos um novo diretório `concept` para armazená-lo, e um novo arquivo `IDEA` para descrevê-lo. E assim, cinco anos e vinte mil revisões se passaram, tal como seria em qualquer história de romance que se preze.

Agora, anos depois, nos questionamos como seria ter de volta o arquivo `IDEA` tal como na revisão 1. Mas o Subversion precisa saber se nós estamos querendo saber sobre como o *atual* arquivo se pareceria na revisão 1, ou se estamos solicitando o conteúdo de qualquer que fosse o arquivo que estivava como `concepts/IDEA` na revisão 1. Certamente estas questões têm respostas diferentes, e devido as revisões marcadoras, é possível obter ambas as respostas. Para ver como o arquivo `IDEA` atual era naquela revisão antiga, você executa:

```
$ svn cat -r 1 concept/IDEA
svn: Unable to find repository location for 'concept/IDEA' in revision 1
```

É claro, neste exemplo, o atual arquivo `IDEA` não existia ainda na revisão 1, então o Subversion lhe exibe um erro. O comando acima é uma versão resumida para uma notação mais longa que relaciona explicitamente uma revisão marcadora. A notação expandida é:

```
$ svn cat -r 1 concept/IDEA@BASE
svn: Unable to find repository location for 'concept/IDEA' in revision 1
```

E quando executada, ela dá os mesmos resultados esperados.

Neste ponto, provavelmente o leitor mais atento está se perguntando se a sintaxe de revisões marcadoras causa problemas em caminhos na c[opia de trabalho ou em URLs que atualmente tenham sinais em si mesmas. Depois de tudo, como o **svn** sabe se `news@11` é o nome de um diretório em minha árvore, ou se é apenas uma sintaxe para a "revisão 11 de `news`"? Felizmente, ainda que o **svn** considere sempre esta última opção, existe uma regra trivial. Você só precisa adicionar um sinal de arroba ao final do caminho, como em `news@11@`. O **svn** só irá se importar com o último sinal de arroba no argumento, e que não seja considerado ilegal omitir um especificador do número da revisão marcadora depois desse arroba. Esta regra também se aplica a caminhos que terminal com um sinal de arroba—você poderia usar `filename@@` para se referir a um arquivo chamado `filename@`.

Vamos considerar a outra questão, então—na revisão 1, como era estava o conteúdo de qualquer que seja o arquivo que estava ocupando o endereço `concepts/IDEA` naquele momento? Vamos usar explicitamente uma revisão marcadora para nos ajudar.

```
$ svn cat concept/IDEA@1
The idea behind this project is to come up with a piece of software
that can frab a naggily wort.  Frabbing naggily worts is tricky
business, and doing it incorrectly can have serious ramifications, so
we need to employ over-the-top input validation and data verification
mechanisms.
```

Perceba que nós não informamos uma revisão operativa neste momento. Isso se deve porque quando uma revisão operativa não é especificada, o Subversion assume como padrão uma revisão operativa que é a mesma da revisão marcadora.

Como você pode ver, a saída da execução de nosso comando parece estar correta. O texto ainda menciona *frabbing naggily worts*, então isto é certamente o arquivo que descreve o software agora chamado de Frabnaggilywort. De fato, podemos verificar isto usando a combinação de uma revisão marcadora e uma revisão operativa. Nós sabemos que em `HEAD`, o projeto Frabnaggilywort está localizado no diretório `frabnaggilywort`. Então nós especificamos que queremos ver como a linha de histórico identificada em `HEAD` como o caminho `frabnaggilywort/IDEA` se parecia na revisão 1.

```
$ svn cat -r 1 frabnaggilywort/IDEA@HEAD
The idea behind this project is to come up with a piece of software
that can frab a naggily wort.  Frabbing naggily worts is tricky
business, and doing it incorrectly can have serious ramifications, so
we need to employ over-the-top input validation and data verification
mechanisms.
```

E as revisões marcadora e operativa nem precisam ser tão triviais. Por exemplo, digamos que `frabnaggilywort` esteja removido na revisão `HEAD`, mas nós sabemos que esse diretório existia na revisão 20, e nós queremos ver as diferenças de seu arquivo `IDEA` entre as revisões 4 e 10. Nós podemos usar a revisão marcadora 20 em conjunto com a URL que deveria conter o diretório `IDEA`

do diretório Frabnaggilywort na revisão 20, e então usar 4 e 10 como nosso intervalo de revisões operativas.

```
$ svn diff -r 4:10 http://svn.red-bean.com/projects/frabnaggilywort/IDEA@20
Index: frabnaggilywort/IDEA
===================================================================
--- frabnaggilywort/IDEA (revision 4)
+++ frabnaggilywort/IDEA (revision 10)
@@ -1,5 +1,5 @@
-The idea behind this project is to come up with a piece of software
-that can frab a naggily wort.  Frabbing naggily worts is tricky
-business, and doing it incorrectly can have serious ramifications, so
-we need to employ over-the-top input validation and data verification
-mechanisms.
+The idea behind this project is to come up with a piece of
+client-server software that can remotely frab a naggily wort.
+Frabbing naggily worts is tricky business, and doing it incorrectly
+can have serious ramifications, so we need to employ over-the-top
+input validation and data verification mechanisms.
```

Felizmente, a maioria das pessoas não se deparam com situações tão complexas desse tipo. Mas se um dia você se deparar, lembre-se que revisões marcadoras são um recurso extra de que o Subversion precisa para reslver ambiguidades.

Modelo de Rede

Em algum momento, será necessário compreender como seu cliente Subversion comunica com seu servidor. A camada de rede do Subversion é abstrata, significando que os clientes Subversion apresentam o mesmo comportamento geral não importando com que tipo de servidor eles estão operando. Seja comunicando no protocolo HTTP (`http://`) com o Servidor HTTP Apache ou comunicando no protocolo personalizado do Subversion (`svn://`) com **svnserve**, o modelo de rede básico é o mesmo. Nesta seção, vamos explicar os princípios básicos deste modelo de rede, incluindo como o Subversion gerencia as questões de autenticação e autorização.

Solicitações e Respostas

O cliente Subversion passa a maior parte de seu tempo gerenciando cópias de trabalho. Quando ele precisa de informações de um repositório remoto, entretanto, ele efetua uma solicitação de rede, e o servidor responde com uma resposta apropriada. Os detalhes do protocolo de rede estão escondidos do usuário—o cliente tenta acessar uma URL, e dependendo do esquema na URL, um protocolo específico é usado para comunicar com o servidor (veja URLs do Repositório).

Execute **svn --version** para ver quais esquemas de URL e protocolos que o cliente sabe como usar.

Quando o processo servidor recebe uma requisição do cliente, ele quase sempre solicita que o cliente se identifique. Ele lança um desafio de autenticação para o cliente, e o cliente responde enviando de volta suas *credenciais* ao servidor. Quando a autenticação for concluída, o servidor responde com a informação original a qual o cliente requisitou. Perceba que este sistema é diferente de sistemas como o CVS em que o cliente oferece credenciais preemptivamente ("efetua um login") ao servidor antes de fazer uma requisição. No Subversion, o servidor é que "pega" as credenciais desafiando o cliente no momento adequado, ao invés de o cliente ter de "inserí-las". Isto torna certas operações mais elegantes. Por exemplo, se um servidor estiver configurado para permitir globalmente que qualquer um leia o repositório, então o servidor nunca vai emitir um desafio de autenticação quando o cliente executar um **svn checkout**.

Se uma dada requisição de rede feita pelo cliente resultar em uma nova revisão sendo criada no repositório (p.ex. **svn commit**), então o Subversion usa o nome de usuário autenticado associado a essas requisições como autor da revisão. Isto é, o nome do usuário autenticado é armazenado como o valor da propriedade `svn:author` na nova revisão (veja “Subversion properties”). Se o cliente não estava autenticado (em outras palavras, se o servidor nunca lançara um desafio de autenticação), então a propriedade `svn:author` será vazia.

Armazenando Credenciais no Cliente

Muitos servidores estão configurados para exigir autenticação em todas solicitações. Isto seria um grande incomodo para os usuários, se eles forem forçados a digitar suas senhas várias vezes. Felizmente, o cliente Subversion possui um remédio para isto—um sistema embutido para armazenamento das credenciais de autenticação em disco. Por padrão, se o cliente de linha de comando responde com sucesso a um desafio de autenticação do servidor, ele salva as credenciais na área privada de configuração de execução do usuário (`~/.subversion/auth/` em sistemas baseado em Unix ou `%APPDATA%/Subversion/auth/` em Windows; veja “Área de Configuração do Tempo de Execução” para maiores detalhes sobre o sistema de configuração de execução). As credenciais aprovadas são armazenadas em disco, chaveadas com uma combinação do nome do servidor, porta, e o domínio de autenticação.

Quando o cliente recebe um desafio de autenticação, ele primeiro procura pelas credenciais apropriadas na cache em disco do usuário. Se aparentemente nenhuma credencial apta está presente, ou se em último caso a credencial armazenada falhar ao autenticar, então o cliente, por padrão, voltará a solicitar ao usuário pela informação necessária.

O leitor consciente de segurança suspeitará imediatamente que há motivo para preocupação aqui. “Armazenar senhas em disco? Isto é terrível! Você nunca deve fazer isto!”

Os desenvolvedores do Subversion reconhecem a legitimidade de tais preocupações, e por esta razão o Subversion trabalha com os mecanismos disponíveis fornecidos pelo sistema e ambiente operacional para tentar minimizar o risco de vazamento destas informações. Aqui está uma descrição de que isto significa para os usuários nas plataformas mais comuns:

- No Windows 2000 e posteriores, o cliente Subversion utiliza os serviços de criptografia padrão do Windows para criptografar a senha no disco. Devido a chave de criptografia ser gerenciada pelo Windows e ser vinculada às credenciais de *login* do próprio usuário, somente o usuário pode descriptografar a senha armazenada. (Note que se a senha da conta Windows do usuário é redefinida por um administrador, todas as senhas armazenadas se tornam indecifráveis. O cliente Subversion se comportará como se elas não existissem, solicitando pelas senhas quando requeridas.)

- Similarmente, no Mac OS X, o cliente Subversion armazena todas as senha de repositório na coleção de chaves de *login* (gerenciada pelo serviço *Keychain*), o qual é protegida pela senha da conta do usuário. As configurações de preferências do usuário podem impor políticas adicionais, como exigir que a senha da conta do usuário seja fornecida cada vez que o Subversion utilize a senha.

- Para outros sistemas operacionais baseado em Unix, nenhum serviço de “keychain” existe. No entanto, a área de armazenamento `auth/` ainda é protegida por permissão para que somente o usuário (proprietário) possa ler dados dela, não todos em geral. As permissões de arquivo do próprio sistema operacional protege as senhas.

Claro que, para o paranóico de verdade, nenhum destes mecanismos satisfaz o teste de perfeição. Então, para aqueles dispostos a sacrificar a conveniência pela segurança extrema, o Subversion oferece vários meios de desabilitar seu sistema de armazenamento de credenciais completamente.

Para desabilitar o armazenamento para um único comando, passe a opção `--no-auth-cache`:

```
$ svn commit -F log_msg.txt --no-auth-cache
Authentication realm: <svn://host.example.com:3690> example realm
```

```
Username:  joe
Password for 'joe':

Adding         newfile
Transmitting file data .
Committed revision 2324.

# password was not cached, so a second commit still prompts us

$ svn delete newfile
$ svn commit -F new_msg.txt
Authentication realm: <svn://host.example.com:3690> example realm
Username:  joe
…
```

Ou, se você quiser desabilitar o armazenamento de credencial permanentemente, você pode editar o arquivo `config` em sua área de configuração do ambiente de execução, e defina a opção `store-auth-creds` para no. Isso evitará o armazenamento de credenciais usadas em qualquer interação que você efetuar com o Subversion no computador afetado. Isso pode ser estendido a todos os usuários no computador, também, ao modificar a área de configuração do sistema como um todo (descrito em “Estrutura da Área de Configuração”).

```
[auth]
store-auth-creds = no
```

Algumas vezes os usuários poderão querer remover credenciais específicas da cache em disco. Para fazer isso, você precisa ir até a área `auth/` e excluir manualmente o arquivo de cache apropriado. As credenciais são armazenadas em arquivos individuais; se você olhar dentro de cada arquivo, você verá chaves e valores. A chave `svn:realmstring` descreve o domínio do servidor específico ao qual o arquivo está associado:

```
$ ls ~/.subversion/auth/svn.simple/
5671adf2865e267db74f09ba6f872c28
3893ed123b39500bca8a0b382839198e
5c3c22968347b390f349ff340196ed39

$ cat ~/.subversion/auth/svn.simple/5671adf2865e267db74f09ba6f872c28

K 8
username
V 3
joe
K 8
password
V 4
blah
K 15
svn:realmstring
V 45
<https://svn.domain.com:443> Joe's repository
END
```

Assim que você localizar o respectivo arquivo de cache, apenas o exclua.

Uma última palavra sobre o comportamento de autenticação do **svn**, especificamente em relação às opções `--username` e `--password`. Muitos dos subcomandos do cliente aceitam estas opções, mas é importante entender que o uso dessas opções *não* envia as credenciais automaticamente ao servidor.

Conforme discutido anteriormente, o servidor "puxa" as credenciais do cliente quando julgar necessário; o cliente não pode "empurrá"-las à vontade. Se um nome de usuário e/ou senha são passados como opções, elas somente serão apresentadas ao servidor se o servidor solicitar elas. [13] Estas opções são normalmente utilizadas para autenticar como um usuário diferente daquele que o Subversion teria optado por padrão (como seu nome de usuário no sistema), ou quando tenta-se evitar as perguntas interativas (como nas chamadas ao comando **svn** a partir de um script).

Aqui está um resumo final que descreve como um cliente Subversion se comporta quando ele recebe um desafio de autenticação.

1. Primeiro, o cliente verifica se o usuário especificou alguma credencial na linda de comando com as opções (`--username` e/ou `--password`). Se não, ou se essas opções não conseguem autenticar com sucesso, então

2. o cliente procura pelo nome, porta e domínio do servidor na área `auth/` do ambiente de execução, para ver se o usuário já possui as credenciais em cache. Se não, ou se as credenciais em cache não conseguem autenticar, então

3. finalmente, o cliente solicita as credenciais ao usuário (a menos que seja instruído a não fazer isso através da opção `--non-interactive` ou suas equivalentes específicas do cliente).

Se o cliente autentica-se com sucesso por qualquer dos métodos listados acima, ele tentará armazenar as credenciais em disco (a menos que o usuário tenha desabilitado este comportamento, como mencionado anteriormente).

[13]Novamente, um erro comum é deixar um servidor mal configurado de forma que ele nunca exija a autenticação do usuário. Quando os usuários passam as opções `--username` e `--password` para o cliente, eles ficam surpresos ao ver que elas nunca foram usadas, ou seja, novas revisões parecem ter sido submetidas anonimamente!

Capítulo 4. Fundir e Ramificar

> “ (É sobre o 'Tronco' que trabalha um cavalheiro.)”
>
> —Confucio

Criar Ramos, Rótulos, e Fundir são conceitos comuns a quase todos os sistemas de controle de Versão. Caso você não esteja familiarizado com estes conceitos, nós oferecemos uma boa introdução a estes nesse capítulo. Se você já conhece estes conceitos, então você vai achar interessante conhecer a maneira como o Subversion os implementa.

Criar Ramos é um item fundamental para Controle de Versão. Se você vai usar o Subversion para gerenciar seus dados, então essa é uma funcionalidade da qual você vai acaber dependendo. Este capítulo assume que você já esteja familiarizado com os conceitos básicos do Subversion(Capítulo 1, *Conceitos Fundamentais*).

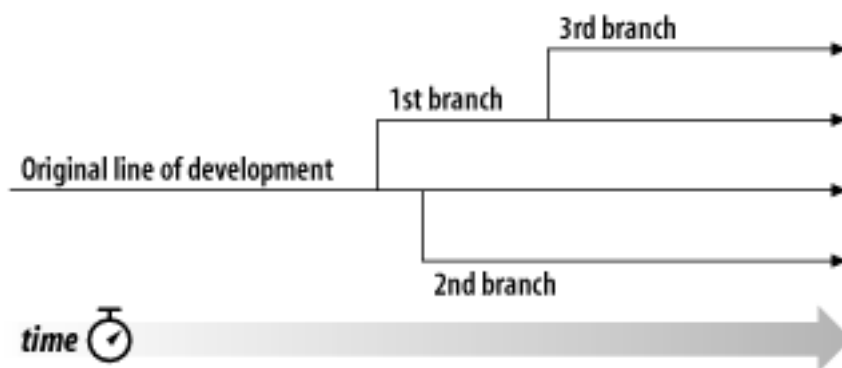
O que é um Ramo?

Suponha que o seu trabalho seja manter um documento de uma divisão de sua empresa, um livro de anotações por exemplo. Um dia, uma outra divisão lhe pede este mesmo livro, mas com alguns “ajustes” para eles, uma vez que eles trabalham de uma forma um pouco diferente.

O que você faz nessa situação? Você faz o óbvio: faz uma segunda cópia do seu documento, e começa a controlar as duas cópias separadamente. Quando cada departamento lhe requisitar alterações, você as realizará em um cópia, ou na outra.

Em raros casos você vai precisar fazer alterações nos dois documentos. Um exemplo, se você encontrar um erro em um dos arquivos, é muito provável que este erro exista na segunda cópia. A final, os dois documentos são quase idênticos, eles têm apenas pequenas diferenças, em locais específicos.

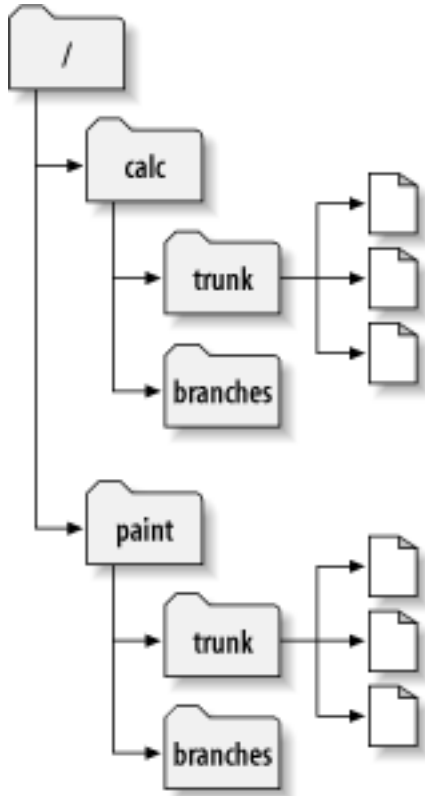
Este é o conceito básico de *Ramo*—isto é, uma linha de desenvolvimento que existe independente de outra linha, e ainda, partilham um histórico em comum, se você olhar para trás na linha tempo. Um Ramo sempre se inicia como cópia de outra coisa, e segue rumo próprio a partir desse ponto, gerando seu próprio histórico. (veja Figura 4.1, “Ramos de desenvolvimento”).

Figura 4.1. Ramos de desenvolvimento

O Subversion tem comandos para ajudar a controlar Ramos paralelos de um arquivo ou diretório. Ele permite você criar ramos copiando seus dados, e ainda lembra que as cópias têm relação entre si. Ainda é possível duplicar cópias de um ramo para outro. Finalmente, ele pode fazer com que partes de sua cópia de trabalho reflitam ramos diferentes, assim você pode “misturar e combinar” diferentes linhas de desenvolvimento no seu trabalho de dia-a-dia.

Usando Ramos

Até aqui, você já deve saber como cada commit cria uma nova árvore de arquivos (chamada de “revisão”) no repositório. Caso não saiba, volte e leia sobre revisões em “Revisões”.

Neste capítulo, vamos usar o mesmo exemplo de antes: Capítulo 1, *Conceitos Fundamentais*. Lembre-se que você e Sally estão compartilhando um repositório que contém dois projetos, `paint` e `calc`. Note que em Figura 4.2, "Layout Inicial do Repositório", entretanto, cada diretório de projeto contém subdiretórios chamados `trunk` e `branches`. O motivo para isso logo ficará mais claro.

Figura 4.2. Layout Inicial do Repositório

Como antes, assuma que você e Sally possuem cópias de trabalho do projeto "calc". Especificamente, cada um de vocês tem uma cópia de trabalho de `/calc/trunk`. Todos os arquivos deste projeto estão nesse diretório ao invés de estarem no `/calc`, porque a sua equipe decidiu que `/calc/trunk` é onde a "Linha Principal" de desenvolvimento vai ficar.

Digamos que você recebeu a tarefa de implementar uma grande funcionalidade nova no projeto. Isso vai requerer muito tempo para escrever, e vai afetar todos os arquivos do projeto. O problema aqui é que você não quer interferir no trabalho de Sally, que está corrigindo pequenos bugs aqui e ali. Ela depende de que a última versão do projeto (em `/calc/trunk`) esteja sempre disponível. Se você começar a fazer commits de suas modificações pouco a pouco, com certeza você vai dificultar o trabalho de Sally.

Um estratégia é "se isolar": você e Sally podem parar de compartilhar informações por uma semana ou duas. Isto é, começar cortar e reorganizar todos os arquivos da sua cópia de trabalho, mas não realizar commit ou update antes de ter terminado todo o trabalho. Existem alguns problemas aqui. Primeiro, não é seguro. A maioria das pessoas gostam de salvar seu trabalho no repositório com frequência, caso algo ruim aconteça por acidente à cópia de trabalho. Segundo, não é nada flexível. Se você faz seu trabalho em computadores diferentes (talvez você tenha uma cópia de trabalho de `/calc/trunk` em duas máquinas diferentes), você terá que, manualmente, copiar suas alterações de uma máquina para outra, ou simplesmente, realizar todo o trabalho em um único computador. Por esse mesmo método, é difícil compartilhar suas constantes modificações com qualquer pessoa. Uma "boa prática" comum em desenvolvimento de software é permitir que outros envolvidos revisem seu trabalho enquanto sendo realizado. Se ninguém verificar seus commits intermediários, você perde um potencial feedback. E por fim, quando você terminar todas as modificações, você pode achar muito difícil fundir seu trabalho com o resto da linha principal de desenvolvimento da empresa. Sally (ou outros) podem ter realizado muitas outras mudanças no repositório que podem ser difíceis de incorporar na sua cópia de trabalho — especialmente se você rodar um **svn update** depois de semanas trabalhando sozinho.

A melhor solução é criar seu próprio ramo, ou linha de desenvolvimento, no repositório. Isso lhe permite salvar seu trabalho ainda incompleto, sem interferir com outros, e ainda você pode escolher que informações compartilhar com seus colaboradores. Você verá exatamente como isso funciona mais à frente.

Criando um Ramo

Criar um ramo é realmente simples— você faz uma cópia do projeto no repositório usando o comando**svn copy**. O Subversion copia não somente arquivos mas também diretórios completos. Neste caso, você quer fazer a cópia do diretório `/calc/trunk`. Onde deve ficar a nova cópia? Onde você quiser— isso depende da "política" do projeto. Digamos que sua equipe tem a política de criar novos ramos na área `/calc/branches` do repositório, e você quer chamar o seu ramo de `my-calc-branch`. Você vai querer criar um novo diretório, `/calc/branches/my-calc-branch`, que inicia sua vida como cópia de `/calc/trunk`.

Há duas maneiras diferentes de fazer uma cópia. Vamos mostrar primeiro a maneira complicada, apenas para deixar claro o conceito. Para começar, faça um checkout do diretório raiz do projeto, `/calc`:

```
$ svn checkout http://svn.example.com/repos/calc bigwc
A  bigwc/trunk/
A  bigwc/trunk/Makefile
A  bigwc/trunk/integer.c
A  bigwc/trunk/button.c
A  bigwc/branches/
Checked out revision 340.
```

Agora para fazer uma cópia basta passar dois caminhos de cópia de trabalho ao comando **svn copy**:

```
$ cd bigwc
$ svn copy trunk branches/my-calc-branch
$ svn status
A  +   branches/my-calc-branch
```

Neste caso, o comando **svn copy** faz uma cópia recursiva do diretório `trunk` para um novo diretório de trabalho, `branches/my-calc-branch`. Como você pode ver pelo comando **svn status**, o novo diretório está agendado para ser adicionado ao repositório. Note também o sinal “+” próximo à letra A. Isso indica o item adicionado é uma *cópia* de algo e não um item novo. Quando você realizar o Commit das modificações, o Subversion vai criar o diretório `/calc/branches/my-calc-branch` no repositório copiando `/calc/trunk`, ao invés de reenviar todos os dados da cópia de trabalho pela rede:

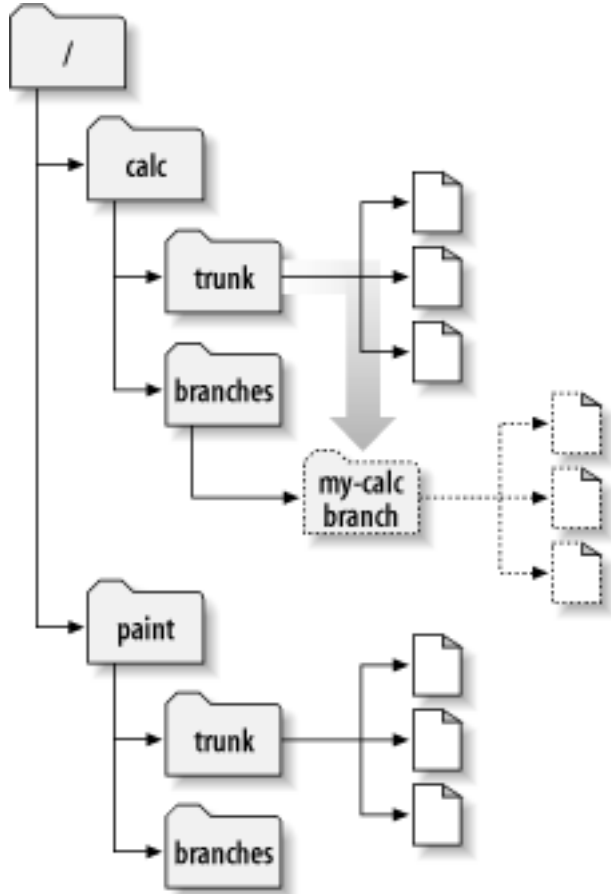
```
$ svn commit -m "Criando um ramo do diretório /calc/trunk."
Adding         branches/my-calc-branch
Committed revision 341.
```

E aqui está o método mais fácil de criar um ramo, o qual nós deveríamos ter lhe mostrado desde o início: o comando **svn copy** é capaz de copiar diretamente duas URLs.

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/branches/my-calc-branch \
      -m "Criando um ramo do diretório /calc/trunk."

Committed revision 341.
```

Do ponto de vista do diretório, não há diferença entre estes dois métodos. Ambos os processos criam um novo diretório na revisão 341, e o novo diretório é uma cópia de `/calc/trunk`. Isso é mostrado em Figura 4.3, “Repositório com uma nova cópia”. Note que o segundo método, entretanto, faz um commit *imediato* em tempo constante. [1] Este é um procedimento mais fácil, uma vez que você não precisa fazer o checkout de uma grande parte do repositório. Na verdade, para usar esta técnica você não precisa se quer ter uma cópia de trabalho. Esta é a maneira que a maioria dos usuários criam ramos.

Figura 4.3. Repositório com uma nova cópia

[1] O Subversion não suporta a cópia entre repositórios distintos. Quando usando URLs com os comandos **svn copy** ou **svn move**, você pode apenas copiar itens dentro de um mesmo repositório.

Cópias Leves

O repositório do Subversion tem um design especial. Quando você copia um diretório, você não precisa se preocupar com o repositório ficando gigante—O Subversion, na realidade, não duplica dados. Ao invés disso, ele cria uma nova entrada de diretório que aponta para uma outra árvore de diretório *já existente*. Caso você seja um usuário Unix, esse é o mesmo conceito do hard-link. Enquanto as modificações são feitas em pastas e arquivos no diretório copiado, o Subversion continua aplicando esse conceito de hard-link enquanto for possível. Os dados somente serão duplicados quando for necessário desambiguizar diferentes versões de um objeto.

É por isso que você quase não vai ouvir os usuários do Subversion reclamando de “Cópias Leves” (*cheap copies*). Não importa o quão grande é o diretório— a cópia sempre será feita em um pequeno e constante espaço de tempo. Na verdade, essa funcionalidade é a base do funcionamento do commit no Subversion: cada revisão é uma “cópia leve” da revisão anterior, com algumas ligeiras modificações em alguns itens.(para ler mais sobre esse assunto, visite o website do Subversion e leia o método “bubble up” nos documentos de design do Subversion.)

Claro que estes mecanismos internos de copiar e compartilhar dados estão escondidos do usuário, que vê apenas cópias das árvores de arquivos. O ponto principal aqui é que as cópias são leves, tanto em tempo quanto em tamanho. Se você criar um ramo inteiro dentro do repositório (usando o comando **svn copy URL1 URL2**), será uma operação rápida, e de tempo constante. Crie ramos sempre que quiser.

Trabalhando com o seu Ramo

Agora que você criou um ramo do projeto, você pode fazer um Checkout para uma nova cópia de trabalho e usá-la.

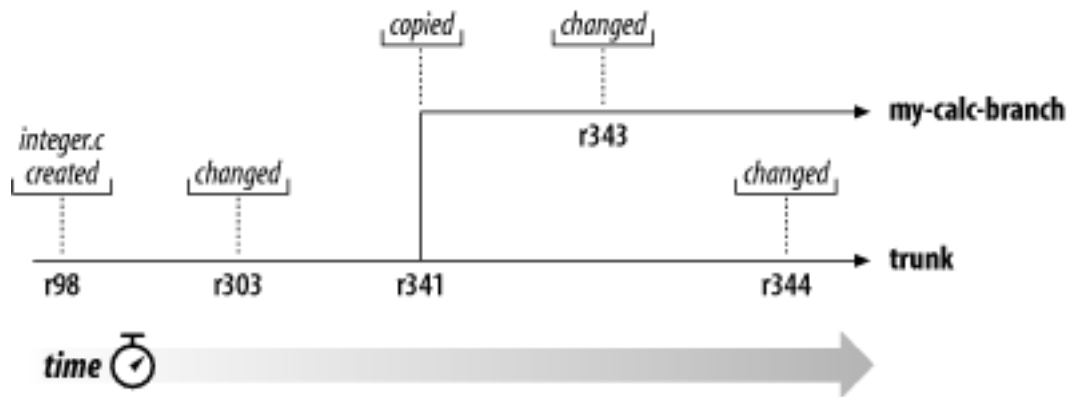
```
$ svn checkout http://svn.example.com/repos/calc/branches/my-calc-branch
A  my-calc-branch/Makefile
A  my-calc-branch/integer.c
A  my-calc-branch/button.c
Checked out revision 341.
```

Não tem nada de especial nessa cópia de trabalho; ela simplesmente aponta para um diretório diferente no repositório. Entretanto, quando você faz o commit de modificações, essas não ficarão visíveis para Sally quando ela fizer Update, porque a cópia de trabalho dela aponta para `/calc/trunk`. (Leia “Atravessando Ramos” logo à frente neste capítulo: o comando **svn switch** é uma forma alternativa de se criar uma cópia de trabalho de um ramo.)

Vamos imaginar que tenha se passado uma semana, e o seguinte commit é realizado:

- Você faz uma modificação em `/calc/branches/my-calc-branch/button.c`, o que cria a revisão 342.
- Você faz uma modificação em `/calc/branches/my-calc-branch/integer.c`, o que cria a revisão 343.
- Sally faz uma modificação em `/calc/trunk/integer.c`, o que cria a revisão 344.

Exitem agora duas linhas independentes de desenvolvimento, mostrando em Figura 4.4, “Ramificação do histórico de um arquivo”, afetando `integer.c`.

Figura 4.4. Ramificação do histórico de um arquivo

As coisas ficam interessantes quando você olha o histórico das alterações feitas na sua cópia de `integer.c`:

```
$ pwd
/home/user/my-calc-branch

$ svn log -v integer.c
------------------------------------------------------------------------
r343 | user | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   M /calc/branches/my-calc-branch/integer.c

* integer.c:  frozzled the wazjub.

------------------------------------------------------------------------
r341 | user | 2002-11-03 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   A /calc/branches/my-calc-branch (from /calc/trunk:340)

Creating a private branch of /calc/trunk.

------------------------------------------------------------------------
r303 | sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  changed a docstring.

------------------------------------------------------------------------
r98 | sally | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  adding this file to the project.

------------------------------------------------------------------------
```

Note que o Subversion está traçando o histórico do seu ramo de `integer.c` pelo tempo, até o momento em que ele foi copiado. Isso mostra o momento em que o ramo foi criado como um evento no histórico, já que `integer.c` foi copiado implicitamente quando `/calc/trunk/` foi copiado. Agora veja o que ocorre quando Sally executa o mesmo comando em sua cópia do arquivo:

```
$ pwd
/home/sally/calc

$ svn log -v integer.c
------------------------------------------------------------------------
r344 | sally | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  fix a bunch of spelling errors.

------------------------------------------------------------------------
r303 | sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  changed a docstring.

------------------------------------------------------------------------
r98 | sally | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  adding this file to the project.

------------------------------------------------------------------------
```

Sally vê suas próprias modificações na revisão 344, e não as modificações que você fez na revisão 343. Até onde o Subversion sabe, esses dois commits afetaram arquivos diferentes em locais distintos no repositório. Entretanto o Subversion *mostra* que os dois arquivos têm um histórico em comum. Antes de ser feita a cópia/ramo na revisão 341, eles eram o mesmo arquivo. É por isso que você e Sally podem ver as alterações feitas nas revisões 303 e 98.

Os conceitos chave por trás de ramos

Há duas lições importantes que você deve se lembrar desta seção. Primeiro, o Subversion não tem um conceito interno de ramos—ele apenas sabe fazer cópias. Quando você copia um diretório, o diretório resultante somente é um “ramo” porque *você* atribui esse significado a ele. Você pode pensar de forma diferente sobre esse diretório, ou tratá-lo de forma diferente, mas para o Subversion é apenas um diretório comum que carrega uma informação extra de histórico. Segundo, devido a este mecanismo de cópia, os ramos no Subversion existem como *diretórios normais do sistema de arquivos* no repositório. Isso é diferente de outros sistemas de controle de versão, onde ramos são criados ao adicionar “rótulos” extra-dimensionais aos arquivos.

Copiando Modificações Entre Ramos

Agora você e Sally estão trabalhando em ramos paralelos do projeto: você está trabalhando no seu próprio ramo, e Sally está trabalhando no *tronco*, ou linha principal de desenvolvimento.

Para projetos que tenham um grande numero de colaboradores, é comum que cada um tenha sua cópia de trabalho do tronco. Sempre que alguem precise fazer uma longa modificação que possa corromper o tronco, o procedimento padrão é criar um ramo privado e fazer os commits neste ramo até que todo o trabalho esteja concluido.

Então, a boa notícia é que você não está interferindo no trabalho de Sally, e vice-versa. A má notícia, é que é muito fácil se *distanciar* do projeto. Lembre-se que um dos problemas com a estratégia do “se isolar” é que quando você terminar de trabalhar no seu ramo, pode ser bem perto de impossível de fundir suas modificações novamente com o tronco do projeto sem um grande numero de conflitos.

Ao invés disso, você e Sally devem continuamente compartilhar as modificações ao longo do seu trabalho. Depende de você para decidir quais modificações devem ser compartilhadas; O Subversion lhe da a capacidade para selecionar o que “copiar” entre os ramos. E quando você terminar de trabalhar no seu ramo, todas as modificações realizadas no seu ramo podem ser copiadas novamente para o tronco.

Copiando modificações específicas

Na seção anterior, nos comentamos que tanto você quanto Sally fizeram alterações em `integer.c` em ramos distintos.Se você olhar a mensagem de log de Sally na revisão 344, você verá que ela corrigiu alguns erros de escrita. Sem duvida alguma, a sua cópia deste arquivo tem os mesmo erros de escrita. É provável que suas futuras modificações a este arquivo vão afetar as mesmas áreas onde foram feitas as correções de escrita, então você tem grandes chances de ter vários conflitos quando for fundir o seu ramo, eventualmente. Portanto, é melhor receber as modificações de Sally agora, *antes* de você começar a trabalhar de forma massiva nessas áreas.

É hora de usar o comando **svn merge**. Esse comando é um primo muito próximo do comando **svn diff** (que você viu em Capítulo 2, *Uso Básico*). Os dois comando comparam dois objetos no repositório e mostram as diferenças. Por exemplo, você pode pedir com o comando **svn diff** para ver com exatidão as mudanças feitas por Sally na revisão 344:

```
$ svn diff -c 344 http://svn.example.com/repos/calc/trunk

Index: integer.c
===================================================================
--- integer.c (revision 343)
+++ integer.c (revision 344)
@@ -147,7 +147,7 @@
     case 6:  sprintf(info->operating_system, "HPFS (OS/2 or NT)"); break;
     case 7:  sprintf(info->operating_system, "Macintosh"); break;
     case 8:  sprintf(info->operating_system, "Z-System"); break;
-    case 9:  sprintf(info->operating_system, "CPM"); break;
+    case 9:  sprintf(info->operating_system, "CP/M"); break;
     case 10:  sprintf(info->operating_system, "TOPS-20"); break;
     case 11:  sprintf(info->operating_system, "NTFS (Windows NT)"); break;
     case 12:  sprintf(info->operating_system, "QDOS"); break;
@@ -164,7 +164,7 @@
     low = (unsigned short) read_byte(gzfile);  /* read LSB */
     high = (unsigned short) read_byte(gzfile); /* read MSB */
     high = high << 8;  /* interpret MSB correctly */
-    total = low + high; /* add them togethe for correct total */
+    total = low + high; /* add them together for correct total */

     info->extra_header = (unsigned char *) my_malloc(total);
     fread(info->extra_header, total, 1, gzfile);
@@ -241,7 +241,7 @@
      Store the offset with ftell() ! */

   if ((info->data_offset = ftell(gzfile))== -1) {
-    printf("error: ftell() retturned -1.\n");
+    printf("error: ftell() returned -1.\n");
     exit(1);
   }

@@ -249,7 +249,7 @@
   printf("I believe start of compressed data is %u\n", info->data_offset);
   #endif
```

```
-  /* Set postion eight bytes from the end of the file. */
+  /* Set position eight bytes from the end of the file. */

   if (fseek(gzfile, -8, SEEK_END)) {
     printf("error: fseek() returned non-zero\n");
```

O comando **svn merge** é quase que o mesmo. Ao invés de imprimir as diferenças no terminal, ele as aplica diretamente à cópia de trabalho classificando como *local modifications*:

```
$ svn merge -c 344 http://svn.example.com/repos/calc/trunk
U  integer.c

$ svn status
M  integer.c
```

A saída do comando **svn merge** mostra a sua cópia de `integer.c` sofreu uma correção. Agora ele contém as modificações feitas por Sally— essas modificações foram "copiadas" do tronco do repositório para a cópia de trabalho do seu ramo privado, e agora existe como uma modificação local. A esta altura, depende de você revisar essa modificação local e ter certeza de funciona.

Em outra simulação, é possível que as coisas não tenham ocorrido tão bem assim, e o arquivo `integer.c` tenha entrado em estado de conflito. Pode ser que você precise resolver o conflito usando procedimentos padrão (veja Capítulo 2, *Uso Básico*), ou se você decidir que fazer a fusão dos arquivos tenha sido uma má idéia, desista e rode o comando **svn revert** para retirar as modificações locais.

Partindo do pressuposto que você revisou as modificações do processo de fusão , então você pode fazer o **svn commit** como de costume. A este ponto, a mudança foi fusionada ao seu ramo no repositório. Em tecnologias de controle de versão, esse ato de copiar mudanças entre ramos recebe o nome de *portar* mudanças.

Quando você fizer o commit das modificações locais, não esqueça de colocar na mensagem de log que você está portando uma modificação especifica de um ramo para outro.Por exemplo:

```
$ svn commit -m "integer.c: ported r344 (spelling fixes) from trunk."
Sending        integer.c
Transmitting file data .
Committed revision 360.
```

Como você verá nas próximas seções, essa é uma "boa pratica" importantíssima a ser seguida.

Porque não usar Patches?

Essa questão pode estar em sua mente, especialmente se você for um usuário de Unix: porque usar o comando **svn merge**? Porque não simplesmente usar o comando do sistema **patch** para realizar esta tarefa? Por exemplo:

```
$ svn diff -c 344 http://svn.example.com/repos/calc/trunk > patchfile
$ patch -p0  < patchfile
Patching file integer.c using Plan A...
Hunk #1 succeeded at 147.
Hunk #2 succeeded at 164.
Hunk #3 succeeded at 241.
Hunk #4 succeeded at 249.
done
```

Neste caso em particular, sim, realmente não há diferença. Mas o comando **svn merge** tem habilidades especiais que superam o comando **patch**. O formato do arquivo usado pelo **patch** é bem limitado; é apenas capaz de mexer o conteúdo dos arquivos. Não há forma de representar mudanças em *arvores*, como o criar, remover e renomear arquivos e diretórios. Tão pouco pode o comando **patch** ver mudanças de propriedades. Se nas modificações de Sally, um diretório tivesse sido criado, a saída do comando **svn diff** não iria fazer menção disso. **svn diff** somente mostra forma limitada do patch, então existem coisa que ele simplesmente não irá mostrar. O comando **svn merge**, por sua vez, pode mostrar modificações em estrutura de árvores e propriedades aplicando estes diretamente em sua cópia de trabalho.

Um aviso: ainda que o comando **svn diff** e o **svn merge** tem conceitos similares, eles apresentam sintaxe diferente em vários casos. Leia sobre isso em Capítulo 9, *Referência Completa do Subversion* para mais detalhes, ou peça ajuda ao comando **svn help**. Por exemplo, o comando **svn merge** precisa de uma cópia de trabalho com destino, isto é, um local onde aplicar as modificações. Se um destino não for especificado, ele assume que você está tentando uma dessas operações:

1. Você quer fundir modificações de diretório no seu diretório de trabalho atual.

2. Você quer fundir as modificações de um arquivo em específico, em outro arquivo de mesmo nome que existe no seu diretório atual de trabalho.

Se você esta fundindo um diretório e não especificou um destino, **svn merge** assume o primeiro caso acima e tenta aplicar as modificações no seu diretório atual. Se você está fundindo um arquivo, e este arquivo (ou arquivo de mesmo nome) existe no diretório atual, o **svn merge** assume o segundo caso, e tenta aplicar as modificações no arquivo local de mesmo nome.

Se você quer que as modificações seja aplicadas em outro local, você vai precisar avisar. Por exemplo, se você está no diretório pai de sua cópia de trabalho, você vai precisar especificar o diretório de destino a receber as modificações:

```
$ svn merge -c 344 http://svn.example.com/repos/calc/trunk my-calc-branch
U   my-calc-branch/integer.c
```

O conceito chave sobre fusão

Agora você viu um exemplo do comando **svn merge**, e você está prestes a ver vários outros. Se você está se sentindo confuso sobre como a fusão funciona, saiba que você não está sozinho. Vários usuários (especial os novos em controle de versão) ficam perplexos com a sintaxe do comando, e sobre como e quando deve ser usado. Mas não temas, esse comando é muito mais simples do que você imagina! Existe uma técnica muito simples para entender exatamente o comportamento do comando **svn merge**.

O principal motivo de confusão é o *nome* do comando. O termo "fundir" de alguma forma denota que se junta ramos, ou que existe uma mistura misteriosa de código ocorrendo. Este não é o caso. O nome mais apropriado para o comando deveria ter sido **svn diff-and-apply** , porque isso é o que acontece: duas árvores de repositório são comparadas, e a diferença é aplicada a uma cópia de trabalho.

O comando recebe três argumentos:

1. Uma árvore de repositório inicial (geralmente chamada de *lado esquerdo* da comparação),

2. Uma árvore de repositório final (geralmente chamada de *lado direito* da comparação),

3. Uma cópia de trabalho para receber as diferenças como modificação local (geralmente chamada de *destino* da fusão).

Uma vez especificados estes três argumentos, as duas árvores são comparadas, e o resultado das diferenças são aplicadas sobre a cópia de trabalho de destino, como modificações locais. Uma vez executado o comando, o resultado não é diferente do que se você tivesse editado manualmente os arquivos, ou rodados vários comandos **svn add** ou **svn delete**. Se você gostar do resultado você pode fazer o commit dele. Se você não gostar do resultado, você pode simplesmente reverter as mudanças com o comando **svn revert**.

A sintaxe do comando **svn merge** lhe permite especificar os três argumentos necessários de forma flexível. Veja aqui alguns exemplos:

```
$ svn merge http://svn.example.com/repos/branch1@150 \
            http://svn.example.com/repos/branch2@212 \
            my-working-copy

$ svn merge -r 100:200 http://svn.example.com/repos/trunk my-working-copy

$ svn merge -r 100:200 http://svn.example.com/repos/trunk
```

A primeira sintaxe usa explicitamente os três argumentos, nomeando cada árvore na forma *URL@REV* e nomeando a cópia de trabalho de destino. A segunda sintaxe pode ser usada como um atalho em situações onde você esteja comparando duas revisões distintas de uma mesma URL. A ultima sintaxe mostra como o argumento da cópia de trabalho de destino é opcional; se omitido, assume como padrão o diretório atual.

Melhores práticas sobre Fusão

Rastreando Fusões manualmente

Fundir modificações parece simples, mas na prática pode se tornar uma dor de cabeça. O problema é que se você repetidamente fundir as modificações de uma ramo com outro, você pode acidentalmente fundir a mesma modificação *duas vezes*. Quando isso ocorre, algumas vezes as coisas vão funcionar corretamente. Quando aplicando um patch em um arquivo, Subversion verifica se o arquivo já possui aquelas modificações e se tiver não faz nada. Mas se a modificações existentes já tiverem modificadas de alguma forma, você terá um conflito.

O ideal seria se o seu sistema de controle de versão prevenisse o aplicar-duas-vezes modificações a um ramo. Ele deveria lembrar automaticamente quais modificações um ramo já recebeu, e ser capaz de listá-los para você. Essa informação deveria ser usada para ajudar a automatizar a Fusão o máximo possivel.

Infelizmente, o Subversion não é esse sistema; ele ainda não grava informações sobre as fusões realizadas. [2] Quando você faz o commit das modificações locais, o repositório não faz a menor idéia se as alterações vieram de um comando **svn merge**, ou de uma edição manual no arquivo.

[2]Entretanto, neste exato momento, essa funcionalidade está sendo preparada!

O que isso significa para você, o usuário? Significa que até que o Subversion tenha essa funcionalidade, você terá que rastrear as informações de Fusão pessoalmente. A melhor maneira de fazer isso é com as mensagens de log do commit. Como mostrado nos exemplos anteriores, é recomendável que sua mensagem de log informe especificamente o número da revisão (ou números das revisões) que serão fundidas ao ramo. Depois, você pode rodar o comando **svn log** para verificar quais modificações o seu ramo já recebeu. Isso vai lhe ajudar a construir um próximo comando **svn merge** que não será redundante com as modificações já aplicadas.

Na próxima seção, vamos mostrar alguns exemplos dessa técnica na prática.

Visualizando Fusões

Primeiro, lembre-se de fundir seus arquivos para a cópia de trabalho quando esta *não* tiver alterações locais e tenha sido atualizada recentemente. Se a sua cópia de trabalho não estiver “limpa”, você pode ter alguns problemas.

Assumindo que a sua cópia de trabalho está no ponto, fazer a fusão não será uma operação de alto risco. Se você não fizer a primeira fusão de forma correta, rode o comando **svn revert** nas modificações e tente novamente.

Se você fez a fusão para uma cópia de trabalho que já possui modificações locais, a mudanças aplicadas pela fusão serão misturadas as pré existentes, e rodar o comando **svn revert** não é mais uma opção. Pode ser impossível de separar os dois grupos de modificações.

Em casos como este, as pessoas se tranquilizam em poder prever e examinar as fusões antes de ocorrerem. Uma maneira simples de fazer isso é rodar o comando **svn diff** com os mesmos argumentos que você quer passar para o comando **svn merge**, como mostramos no primeiro exemplo de fusão. Outro método de prever os impactos é passar a opção `--dry-run` para o comando de fusão:

```
$ svn merge --dry-run -c 344 http://svn.example.com/repos/calc/trunk
U  integer.c

$ svn status
#  nothing printed, working copy is still unchanged.
```

A opção `--dry-run` não aplica qualquer mudança para a copia de trabalho. Essa opção apenas exibe os códigos que *seriam* escritos em uma situação real de fusão. É útil poder ter uma previsão de “auto nível” da potencial fusão, para aqueles momentos em que o comando **svn diff** dá detalhes até demais.

Fundir conflitos

Assim como no comando **svn update**, o comando **svn merge** aplica modificações à sua cópia de trabalho. E portanto também é capaz de criar conflitos. Entretanto, os conflitos criados pelo comando **svn merge** são um tanto diferentes, e essa seção explica essas diferenças.

Para começar, assuma que sua cópia de trabalho não teve modificações locais. Quando você faz a atualização com o comando **svn update** para um revisão específica, as modificações enviadas pelo servidor vão ser sempre aplicadas à sua cópia de trabalho “sem erros”. O servidor produz o delta a partir da comparação de duas árvores: uma imagem virtual de sua cópia de trabalho, e a árvore da revisão na qual está interessado. Como o lado esquerdo da comparação é exatamente igual ao que você já possui, é garantido que o delta converterá corretamente sua cópia de trabalho, para a revisão escolhida no lado direito da compração.

Entretanto, o comando **svn merge** não possui essa garantia e pode ser bem mais caótico: o usuário pode pedir ao servidor para comparar *qualquer* árvore, até mesmo árvores que não tenham relação com a sua cópia de trabalho! Isso significa que existem uma grande margem para erro humano. Usuário vão acabar por compara duas árvores erradas, criando um delta que não se aplica sem conflitos. O

comando **svn merge** vai fazer o melhor possível para aplicar o delta o máximo possível, mas em algumas partes isso pode ser impossível. Assim como no comando Unix **patch** que as vezes reclama sobre "failed hunks", o **svn merge** vai reclamar sobre "alvos perdidos":

```
$ svn merge -r 1288:1351 http://svn.example.com/repos/branch
U  foo.c
U  bar.c
Skipped missing target: 'baz.c'
U  glub.c
C  glorb.h

$
```

O exemplo anterior pode ser um caso no qual o arquivo `baz.c` existe nas duas imagens dos ramos que estão sendo comparados, e o delta resultante quer modificar o conteúdo do arquivo, mas o arquivo não existe na cópia de trabalho. Independente do caso, a mensagem de "skipped" significa que o usuário está, muito provavelmente, comparando árvores incorretas; esse é o sinal clássico de erro do usuário. Quando isso acontece, é fácil reverter recursivamente as modificações criadas pela fusão (**svn revert --recursive**), delete qualquer arquivo não versionado deixado pelo revert, e rode novamente o comando **svn merge** usando outros argumentos.

Note também que o exemplo anterior mostra um conflito no arquivo `glorb.h`. Nós já mostramos que a cópia local não possui modificações:como um conflito pôde acontecer? Novamente, uma vez que o usuário pode usar o comando **svn merge** para definir e aplicar qualquer delta antigo para a cópia de trabalho, o delta pode conter alterações que não se aplicam sem erros ao arquivo local, mesmo que o arquivo não tenha modificações locais.

Outra pequena diferença entre os comandos **svn update** e **svn merge** é o nome dos arquivos de texto criados quando ocorre um conflito. Em "Resolvendo Conflitos (Combinando Alterações de Outros)", vimos que um update produz arquivos nomeados de `filename.mine`, `filename.rOLDREV`, e `filename.rNEWREV`. Entretanto, quando o comando **svn merge** produz um conflito, ele cria três arquivos nomeados como `filename.working`, `filename.left`, e `filename.right`. Neste caso, os termos "left" e "right" estão indicando de que lado da comparação vieram os arquivos. Em todo caso, esses nomes vão ajuda-lo a diferenciar conflitos que são resultado de um update ou de uma fusão.

Percebendo ou Ignorando os Ancestrais

Ao conversar com um desenvolvedor do Subversion, você frequentemente ouviria referências ao termo *ancestral*. Esta palavra é usada para descrever a relação entre dois objetos em um repositório: se estiverem relacionados entre si, então um objeto é dito ser um ancestral do outro.

Por exemplo, suponha que você submeta a revisão 100, a qual inclui uma mudança num arquivo `foo.c`. Então, `foo.c@99` é o ancestral de "ancestral" de `foo.c@100`. Por outro lado, suponha que você submeta a exclusão do arquivo `foo.c` na revisão 101, e então adicione um novo arquivo com o mesmo nome na revisão 102. Neste caso, `foo.c@99` e `foo.c@102` podem parecer estar relacionados (afinal, eles têm o mesmo caminho), mas de fato eles são objetos completamente diferentes no repositório. Eles não compartilham histórico ou "ancestralidade".

A razão para abordar isto é destacar uma importante diferença entre **svn diff** e **svn merge**. O primeiro comando ignora a ancestralidade, enquanto que este último é bastante sensível a ela. Por exemplo, se você solicitar que o **svn diff** compare as revisões 99 e 102 do arquivo `foo.c`, você deveria ver diferenças em termos de linhas do arquivo em cada revisão; o comando `diff` é cego ao comparar dois caminhos. Mas se você solicitar ao **svn merge** para comparar os mesmos dois objetos, o subcomando deve perceber que estes dois objetos não estão relacionados e primeiro tentará excluir o arquivo antigo, e então adicionar o arquivo novo; a saída deveria indicar uma exclusão seguida por uma adição:

```
D  foo.c
```

```
A  foo.c
```

A maioria das fusões envolve comparação de árvores ancestralmente relacionadas umas às outras, e assim o **svn merge** por padrão possui este comportamento. Ocasionalmente, no entanto, você pode querer que o comando `merge` compare duas árvores não relacionadas. Por exemplo, você pode ter importado duas árvores de código-fonte representando distribuições de diferentes fornecedores de um projeto de software (veja “Ramos de fornecedores”). Se você solicitar que o **svn merge** compare as duas árvores, você deveria ver a exclusão da primeira árvore inteira, seguida da adição da segunda árvore inteira! Nessas situações, você vai querer que o **svn merge** faça uma comparação baseada apenas em caminhos, ignorando quaisquer relações entre arquivos e diretórios. Adicione a opção `--ignore-ancestry` a seu comando merge, e ele se comportará como o **svn diff**. (E reversalmente, a opção `--notice-ancestry` fará com que o **svn diff** se comporte como o comando `merge`.)

Fusões e Movimentações

Um desejo comum é refatorar código-fonte, especialmente em projetos de software na linguagem Java. Arquivos e diretórios são mexidos e renomeados, possivelmente provocando transtornos a todos que estiverem trabalhando no projeto. Parece um caso perfeito para criar um ramo, não? Apenas crie um ramo, modifique as coisas inteiramente, e então mescle o ramo de volta ao tronco principal, certo?

Infelizmente, no momento este cenário não funciona tão bem, sendo algo considerado como um dos pontos fracos do Subversion. O problema é que o comando **update** do Subversion não é tão robusto quanto poderia ser, especialmente ao lidar com operações de cópia e movimentações.

Quando você usa o **svn copy** para duplicar um arquivo, o repositório se lembra de onde o novo arquivo veio, mas falha ao transmitir essa informação para o cliente que está executando um **svn update** ou um **svn merge**. Ao invés de dizer para o cliente, “Copie este arquivo que você já possui para este novo local”, ele envia informação acerca de um arquivo completamente novo. Isto pode levar a problemas, especialmente pelo fato de que a mesma coisa acontece com arquivos renomeados. Um fato pouco conhecido pouco conhecido sobre o Subversion é que ainda lhe falta um recurso para “renomeação efetiva”—o comando **svn move** nada mais é que uma combinação de **svn copy** e **svn delete**.

Por exemplo, suponha que ao trabalhar em seu ramo particular, você renomeie `integer.c` para `whole.c`. Efetivamente você criou um novo arquivo em seu ramo que é uma cópia do arquivo original e excluiu o arquivo original. Enquanto isso, de volta ao `trunk`, Sally submeteu algumas melhorias em `integer.c`. Agora você decide mesclar seu ramo ao tronco:

```
$ cd calc/trunk

$ svn merge -r 341:405 http://svn.example.com/repos/calc/branches/my-calc-branch
D   integer.c
A   whole.c
```

À primeira vista, isto não parece tão ruim, mas provavelmente também não era o que você ou Sally esperavam. A operação de mesclagem excluiu a última versão do arquivo `integer.c` (aquela que continha as últimas alterações de Sally), e adicionou cegamente seu novo arquivo `whole.c`—que é uma duplicata da versão *mais antiga* de `integer.c`. O efeito em cascata é que mesclar sua “renomeação” no ramo removeu as modificações recentes de Sally para a última revisão!

Mas isto não é uma perda de dados real; as modificações de Sally ainda estão no histórico do repositório, mas o que de fato aconteceu pode não ser óbvio de imediato. A moral dessa história é que até que o Subversion evolua, tenha cuidado ao mesclar cópias e renomeações a partir de um ramo para outro.

Casos Comuns de Utilização

Há muitos usos diferentes para ramificações e para o **svn merge**, e esta seção descreve os usos mais comuns com os quais você provavelmente irá se deparar.

Mesclando um Ramo Inteiro para Outro

Para completar nosso exemplo de execução, vamos avançar no tempo. Suponha que vários dias tenham se passado, e que muitas alterações tenham acontecido tanto no tronco quanto em seu ramo particular. Suponha que você tenha terminado de trabalhar seu ramo particular; e que o recurso ou correção de bug tenha finalmente terminado, e que agora você quer mesclar todas as modificações de seu ramo de volta para o tronco principal para que os outros usufruam.

Então como usamos o **svn merge** neste cenário? Lembre-se de que este comando compara duas árvores, e aplica as diferenças em uma cópia de trabalho. Então para receber as modificações, você precisa ter uma cópia de trabalho do tronco. Vamos assumir que você ainda possua uma cópia original (completamente atualizada), ou que você recentemente tenha obtido uma nova cópia de trabalho de `/calc/trunk`.

Mas quais duas árvores deveriam ser comparadas? À primeira vista a resposta pode parecer óbvia: apenas compare a árvore mais recente do tronco com sua árvore mais recente de seu ramo. Mas cuidado—esta suposição está *errada*, e isso costuma confundir muito os novos usuários! Como o **svn merge** opera como o **svn diff**, comparar as últimas versões das árvores do tronco e do ramo *não* descreve apenas o conjunto de modificações que você fez em seu ramo. Tal comparação exibe muito mais mudanças: ele não apenas exibe o efeito das modificações de seu ramo, mas também todas as alterações de *removação* que nunca aconteceram em seu ramo.

Para expressar apenas as modificações que aconteceram em seu ramo, você precisa comparar o estado inicial de seu ramo com seu estado final. Usando um **svn log** em seu ramo, você pode ver que seu ramo foi criado na revisão 341. E o estado final de seu ramo é simplesmente uma dada forma de uso da revisão `HEAD`. Isso significa que você deve comparar as revisões 341 e `HEAD` do seu diretório branch, e aplicar estas diferenças na cópia de trabalho de trunk.

Uma ótima maneira de encontrar a revisão na qual um ramo foi criado (a “base” do ramo) é usar a opção `--stop-on-copy` do comando **svn log**. O subcomando log normalmente irá mostrar cada modificação feita no ramo, incluindo o rastreamento de volta além da operação de cópia que criou o ramo. Então, normalmente, você irá ver o histórico do tronco também. A opção `--stop-on-copy` irá parar a saída do log assim que o **svn log** detecte que seu alvo foi copiado ou renomeado.

Assim, no caso de nosso exemplo,

```
$ svn log -v --stop-on-copy \
          http://svn.example.com/repos/calc/branches/my-calc-branch
…
------------------------------------------------------------------------
r341 | user | 2002-11-03 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   A /calc/branches/my-calc-branch (from /calc/trunk:340)

$
```

Como esperado, a última revisão exibida por este comando é é a revisão na qual o ramo `my-calc-branch` foi criado por cópia.

E então, aqui está o último procedimento para mesclagem:

```
$ cd calc/trunk
$ svn update
At revision 405.
```

```
$ svn merge -r 341:405 http://svn.example.com/repos/calc/branches/my-calc-branch
U   integer.c
U   button.c
U   Makefile

$ svn status
M   integer.c
M   button.c
M   Makefile

# ...examine os diffs, compilações, testes, etc...

$ svn commit -m "Merged my-calc-branch changes r341:405 into the trunk."
Sending        integer.c
Sending        button.c
Sending        Makefile
Transmitting file data ...
Committed revision 406.
```

Novamente, perceba que a mensagem de log do commit menciona bem especificamente o intervalo de modificações que foram mescladas para o tronco. Sempre se lembre de fazer isso, pois é uma informação crítica de que você irá precisar depois.

Por exemplo, suponha que você decida continuar trabalhando em seu ramo por mais uma semana, para concluir uma melhoria em seu recurso original ou uma correção de bug. A revisão HEAD do repositório agora é a 480, e você está pronto para fazer outra mesclagem de seu ramo particular com o tronco principal. Mas como já discutido em “Melhores práticas sobre Fusão”, você não quer mesclar as modificações que você já mesclou anteriormente; o que você quer é mesclar todas as coisas “novas” em seu ramo desde a última mesclagem que você fez. O truque é conferir exatamente quais são as coisas novas.

O primeiro passo é executar **svn log** no tronco, e procurar por uma mensagem de log da última vez que você mesclou um ramo:

```
$ cd calc/trunk
$ svn log
…
------------------------------------------------------------------------
r406 | user | 2004-02-08 11:17:26 -0600 (Sun, 08 Feb 2004) | 1 line

Merged my-calc-branch changes r341:405 into the trunk.
------------------------------------------------------------------------
…
```

Aha! Como todas as modificações no ramo que aconteceram entre as revisões 341 e 408 já foram previamente mescladas para o tronco gerando a revisão 406, você agora sabe que deve mesclar apenas as alterações feitas depois disso—comparando as revisões HEAD.

```
$ cd calc/trunk
$ svn update
At revision 480.

# Percebemos que atualmente HEAD está em 480, então usamos isso para fazer a mesclagem:

$ svn merge -r 406:480 http://svn.example.com/repos/calc/branches/my-calc-branch
U   integer.c
U   button.c
```

```
U   Makefile

$ svn commit -m "Merged my-calc-branch changes r406:480 into the trunk."
Sending        integer.c
Sending        button.c
Sending        Makefile
Transmitting file data ...
Committed revision 481.
```

Agora o tronco contém a segunda leva completa de modificações feitas no ramo. Neste ponto, você pode tanto excluir o seu ramo (falaremos mais sobre isso posteriormente), ou continuar trabalhando em seu ramo e repetir este procedimento para mesclagens subsequentes.

Desfazendo Alterações

Outro uso comum do **svn merge** é para desfazer uma modificação que já foi submetida ao repositório. Suponha que você esteja trabalhando alegremente na cópia de trabalho de `/calc/trunk`, e você descobre que a modificação que havia sido feita na revisão 303, que modificou o arquivo `integer.c`, está completamente errada. E que ela nunca deveria ter acontecido, nem tampouco submetida. Você pode usar o **svn merge** para "desfazer" a modificação em cópia de trabalho, e então submeter a modificação local para o repositório. Tudo o que você precisa fazer é especificar uma diferença *reversa*. (Você pode fazer isto especificando `--revision 303:302`, ou também o equivalente `--change -303`.)

```
$ svn merge -c -303 http://svn.example.com/repos/calc/trunk
U  integer.c

$ svn status
M  integer.c

$ svn diff
…
# verify that the change is removed
…

$ svn commit -m "Undoing change committed in r303."
Sending        integer.c
Transmitting file data .
Committed revision 350.
```

Uma maneira de pensar o repositório é como um grupo específico de modificações (alguns sistemas de controle de versão chamam a isto de *conjuntos de mudanças* ou *changesets*). Usando a opção `-r`, você pode solicitar que o **svn merge** aplique um conjunto de mudanças, ou um intervalo inteiro de conjuntos de mudanças, à sua cópia de trabalho. Em nosso caso em questão, como queremos desfazer uma mudança, estamos solicitando que o **svn merge** aplique o conjunto de mudanças #303 *retrospectivamente* de volta à nossa cópia de trabalho.

Subversion e os Conjuntos de Mudanças

Cada um parece ter uma definição ligeiramente diferente do que seja um “conjunto de mudanças”, ou ao menos diferentes expectativas sobre o que significa um sistema de controle de versão possuir “recursos para lidar com conjuntos de mudanças”. Para nosso propósito, digamos que um conjunto de mudança seja apenas uma porção de alterações associadas a um nome único. As alterações podem incluir modificações textuais ao conteúdo de arquivos, mudanças em uma estrutura de árvore, ou ajustes em metadados. Falando de uma forma mais geral, um conjunto de mudanças é apenas um *patch* com um nome a partir do qual você pode se referir.

No Subversion, um número global de revisão N nomeia uma árvore no repositório: é a forma como o repositório se parece após a N-ésima submissão. É também o nome de um conjunto de mudanças implícito: se você compara a árvore N com a árvore N-1, você pode derivar o patch exato que foi submetido. Por esta razão, é fácil pensar que a “revisão N” não é apenas uma árvore, mas um conjunto de mudanças também. Se você usar algum sistema de tíquetes (ou *issue tracker*) para gerenciar bugs, você pode usar os números de revisão para se referir a patches específicos que corrigem determinados bugs—por exemplo, “a demanda deste tíquete foi corrigida na revisão 9238.”. Alguém pode então executar **svn log -r9238** para ler exatamente sobre o conjunto de mudanças que corrigiram o bug, e executar **svn diff -c 9238** para ver a correção em si. E o comando `merge` do Subversion também usa números de revisão. Você pode mesclar conjuntos de mudança específicos a partir de um ramo para outro discriminando-os nos argumentos do comando merge: **svn merge -r9237:9238** deve incorporar o conjunto de mudanças #9238 à sua cópia de trabalho.

Tenha em mente que voltar uma mudança como neste caso é uma operação de **svn merge** como outra qualquer, então você deveria usar **svn status** e **svn diff** para confirmar que seu trabalho esteja no estado em que você quer que esteja, e então usar **svn commit** para enviar a versão final para o repositório. Depois de submetido, este conjunto de mudanças em particular não estará mais refletido na revisão `HEAD`.

Novamente, você pode estar pensando: bem, isto não desfaz exatamente a submissão, não é? A modificação ainda existe na revisão 303. Se alguém obtiver uma versão do projeto `calc` entre as revisões 303 e 349, elas ainda conterão a tal modificação incorreta, certo?

Sim, isto é verdade. Quando nós falamos sobre “remover” uma modificação, estávamos realmente falando sobre removê-la da revisão `HEAD`. A modificação original ainda existirá no histórico do repositório. Na maioria das situações, isto é o suficiente. Afinal, a maioria das pessoas estão apenas interessadas em rastrear a revisão `HEAD` de um projeto. Porém, há alguns casos especiais onde você realmente pode querer destruir todas as evidências da submissão errônea. (Talvez alguém submetido acidentalmente um documento confidencial.) Isto não é tão fácil de se fazer, pois o Subversion foi desenvolvido deliberadamente para nunca perder informação. As revisões são árvores imutáveis as quais são construídas umas a partir das outras. Remover uma revisão do histórico deveria causar um efeito dominó, criando o caos em todas as revisões subsequentes e possivelmente invalidando todas as cópias de trabalho. [3]

Ressucitando Itens Excluídos

O grande ponto sobre sistemas de controle de versão é que a informação nunca é perdida. Mesmo quando você exclui um arquivo ou diretório, ele pode até não estar mais presente na revisão `HEAD`, mas o objeto ainda existe nas revisões mais antigas. Uma das questões mais comuns que novos usuários se perguntam é, “Como eu faço para obter meu arquivo ou diretório antigo de volta?”.

O primeiro passo é definir exatamente **qual** ítem você está tentando ressucitar. Aqui há uma metáfora útil: você pode pensar como se cada objeto no repositório existisse em uma espécie de sistema bi-dimensional. A primeira coordenada é uma determinada árvore de revisão, e a segunda coordenada

[3]Entretanto, o projeto Subversion tem planos de, algum dia, implementar um comando que possa cumprir a tarefa de excluir permanentemente alguma informação. Enquanto isso, veja “svndumpfilter” para uma possível solução.

é o caminho dentro daquela árvore. Assim cada versão de seu arquivo ou diretório pode ser definida por um dado par de coordenadas. (Lembre-se da sintaxe de “revisões marcadoras”—foo.c@224—apresentada em “Revisões Marcadoras e Revisões Operativas”.)

Primeiramente, você pode precisar usar um **svn log** para descobrir o par de coordenadas exato que você quer ressucitar. Uma boa estratégia é executar **svn log --verbose** em um diretório onde seu item excluído costumava estar. A opção `--verbose (-v)` exibe uma lista de todos os itens que mudaram em cada revisão; tudo que você precisa fazer é encontrar a revisão na qual você excluir o arquivo ou diretório. Você pode fazer isto visualmente, ou usando outra ferramenta para examinar a saída dos registros de log (usando **grep**, ou talvez com uma busca incremental em um editor).

```
$ cd parent-dir
$ svn log -v
…
------------------------------------------------------------------------
r808 | joe | 2003-12-26 14:29:40 -0600 (Fri, 26 Dec 2003) | 3 lines
Changed paths:
   D /calc/trunk/real.c
   M /calc/trunk/integer.c

Added fast fourier transform functions to integer.c.
Removed real.c because code now in double.c.
…
```

No exemplo, estamos assumindo que você está procurando um arquivo excluído chamado `real.c`. Olhando os logs de um diretório-pai, você percebeu que este arquivo foi excluído na revisão 808. Portanto, a última versão do arquivo existia na revisão imediatamente anterior a essa. Conclusão: você quer ressucitar o caminho `/calc/trunk/real.c` a partir da revisão 807.

Esta foi a parte difícil—a pesquisa. Agora que você sabe o que você quer restaurar, você tem duas diferentes escolhas.

Uma opção é usar **svn merge** para aplicar a revisão 808 “ao contrário”. (Nós já falamos sobre como desfazer modificações, veja “Desfazendo Alterações”.) Isto teria o efeito de re-adicionar o arquivo `real.c` como uma modificação local. O arquivo deveria ser agendado para adição, e após ser submetido, o arquivo deve estar novamente presente na revisão `HEAD`.

Neste exemplo em particular, no entanto, esta provavelmente não é a melhor estratégia. A aplicação reversa da revisão 808 não apenas agenda `real.c` para adição, mas a mensagem de log indica que ele também deve desfazer certas alterações em `integer.c`, o que você não quer. Certamente, você poderia fazer uma mesclagem reversa da revisão 808 e então executar um **svn revert** nas modificações locais em `integer.c`, mas esta técnica não é bem escalável. E se tivéssemos 90 arquivos modificados na revisão 808?

Uma segunda, e mais precisa estratégia envolve não usar o **svn merge**, mas o comando **svn copy** em seu lugar. Simplesmente copie a revisão exata e o caminho como “par de coordenadas” do repositório para sua cópia de trabalho:

```
$ svn copy -r 807 \
           http://svn.example.com/repos/calc/trunk/real.c ./real.c

$ svn status
A  +   real.c

$ svn commit -m "Resurrected real.c from revision 807, /calc/trunk/real.c."
Adding         real.c
Transmitting file data .
```

```
Committed revision 1390.
```

O sinal de mais na saída do comando status indica que o item não está meramente agendado para adição, mas agendado para adição “com histórico”. O Subversion lembra de onde ele foi copiado. No futuro, executar **svn log** neste arquivo irá percorrer até o arquivo ressucitado e através do histórico que ele tinha antes da revisão 807. Em outras palavras, este novo `real.c` não é realmente novo; é um descendente direto do arquivo original que fora excluído.

Apesar de nosso exemplo nos mostrar uma ressurreição de arquivo, veja que estas mesmas técnicas funcionam muito bem também para ressucitar diretórios excluídos.

Common Branching Patterns

Version control is most often used for software development, so here's a quick peek at two of the most common branching/merging patterns used by teams of programmers. If you're not using Subversion for software development, feel free to skip this section. If you're a software developer using version control for the first time, pay close attention, as these patterns are often considered best practices by experienced folk. These processes aren't specific to Subversion; they're applicable to any version control system. Still, it may help to see them described in Subversion terms.

Release Branches

Most software has a typical lifecycle: code, test, release, repeat. There are two problems with this process. First, developers need to keep writing new features while quality-assurance teams take time to test supposedly-stable versions of the software. New work cannot halt while the software is tested. Second, the team almost always needs to support older, released versions of software; if a bug is discovered in the latest code, it most likely exists in released versions as well, and customers will want to get that bugfix without having to wait for a major new release.

Here's where version control can help. The typical procedure looks like this:

- *Developers commit all new work to the trunk.* Day-to-day changes are committed to `/trunk`: new features, bugfixes, and so on.
- *The trunk is copied to a “release” branch.* When the team thinks the software is ready for release (say, a 1.0 release), then `/trunk` might be copied to `/branches/1.0`.
- *Teams continue to work in parallel.* One team begins rigorous testing of the release branch, while another team continues new work (say, for version 2.0) on `/trunk`. If bugs are discovered in either location, fixes are ported back and forth as necessary. At some point, however, even that process stops. The branch is “frozen” for final testing right before a release.
- *The branch is tagged and released.* When testing is complete, `/branches/1.0` is copied to `/tags/1.0.0` as a reference snapshot. The tag is packaged and released to customers.
- *The branch is maintained over time.* While work continues on `/trunk` for version 2.0, bugfixes continue to be ported from `/trunk` to `/branches/1.0`. When enough bugfixes have accumulated, management may decide to do a 1.0.1 release: `/branches/1.0` is copied to `/tags/1.0.1`, and the tag is packaged and released.

This entire process repeats as the software matures: when the 2.0 work is complete, a new 2.0 release branch is created, tested, tagged, and eventually released. After some years, the repository ends up with a number of release branches in “maintenance” mode, and a number of tags representing final shipped versions.

Feature Branches

A *feature branch* is the sort of branch that's been the dominant example in this chapter, the one you've been working on while Sally continues to work on `/trunk`. It's a temporary branch created to work on

a complex change without interfering with the stability of `/trunk`. Unlike release branches (which may need to be supported forever), feature branches are born, used for a while, merged back to the trunk, then ultimately deleted. They have a finite span of usefulness.

Again, project policies vary widely concerning exactly when it's appropriate to create a feature branch. Some projects never use feature branches at all: commits to `/trunk` are a free-for-all. The advantage to this system is that it's simple—nobody needs to learn about branching or merging. The disadvantage is that the trunk code is often unstable or unusable. Other projects use branches to an extreme: no change is *ever* committed to the trunk directly. Even the most trivial changes are created on a short-lived branch, carefully reviewed and merged to the trunk. Then the branch is deleted. This system guarantees an exceptionally stable and usable trunk at all times, but at the cost of tremendous process overhead.

Most projects take a middle-of-the-road approach. They commonly insist that `/trunk` compile and pass regression tests at all times. A feature branch is only required when a change requires a large number of destabilizing commits. A good rule of thumb is to ask this question: if the developer worked for days in isolation and then committed the large change all at once (so that `/trunk` were never destabilized), would it be too large a change to review? If the answer to that question is “yes”, then the change should be developed on a feature branch. As the developer commits incremental changes to the branch, they can be easily reviewed by peers.

Finally, there's the issue of how to best keep a feature branch in “sync” with the trunk as work progresses. As we mentioned earlier, there's a great risk to working on a branch for weeks or months; trunk changes may continue to pour in, to the point where the two lines of development differ so greatly that it may become a nightmare trying to merge the branch back to the trunk.

This situation is best avoided by regularly merging trunk changes to the branch. Make up a policy: once a week, merge the last week's worth of trunk changes to the branch. Take care when doing this; the merging needs to be hand-tracked to avoid the problem of repeated merges (as described in “Rastreando Fusões manualmente”). You'll need to write careful log messages detailing exactly which revision ranges have been merged already (as demonstrated in “Mesclando um Ramo Inteiro para Outro”). It may sound intimidating, but it's actually pretty easy to do.

At some point, you'll be ready to merge the “synchronized” feature branch back to the trunk. To do this, begin by doing a final merge of the latest trunk changes to the branch. When that's done, the latest versions of branch and trunk will be absolutely identical except for your branch changes. So in this special case, you would merge by comparing the branch with the trunk:

```
$ cd trunk-working-copy

$ svn update
At revision 1910.

$ svn merge http://svn.example.com/repos/calc/trunk@1910 \
            http://svn.example.com/repos/calc/branches/mybranch@1910
U  real.c
U  integer.c
A  newdirectory
A  newdirectory/newfile
…
```

By comparing the `HEAD` revision of the trunk with the `HEAD` revision of the branch, you're defining a delta that describes only the changes you made to the branch; both lines of development already have all of the trunk changes.

Another way of thinking about this pattern is that your weekly sync of trunk to branch is analogous to running **svn update** in a working copy, while the final merge step is analogous to running **svn commit** from a working copy. After all, what else *is* a working copy but a very shallow private branch? It's a branch that's only capable of storing one change at a time.

Atravessando Ramos

O comando **svn switch** transforma uma cópia de trabalho existente para refletir um ramo diferente. Enquanto este comando não é estritamente necessário para trabalhar com ramos, ele oferece um bom atalho. Em nosso exemplo anterior, depois de criar seu ramo pessoal, você obteve uma cópia de trabalho atualizada do novo diretório do repositório. Em vez disso, você pode simplesmente pedir ao Subversion que mude sua cópia de trabalho de `/calc/trunk` para espelhar o local do novo ramo:

```
$ cd calc

$ svn info | grep URL
URL: http://svn.example.com/repos/calc/trunk

$ svn switch http://svn.example.com/repos/calc/branches/my-calc-branch
U   integer.c
U   button.c
U   Makefile
Updated to revision 341.

$ svn info | grep URL
URL: http://svn.example.com/repos/calc/branches/my-calc-branch
```

Depois da “comutação” para o ramo, sua cópia de trabalho não é diferente daquilo que você obteria fazendo uma cópia atualizada do diretório. E ainda é usualmente mais eficiente usar este comando, porque muitas vezes os ramos diferem somente em poucos detalhes. O servidor envia somente o conjunto mínimo de mudanças necessárias para fazer sua cópia de trabalho refletir o diretório do ramo.

O comando **svn switch** também possui uma opção `--revision` (`-r`), assim você não precisa sempre mover sua cópia de trabalho para a revisão `HEAD` do ramo.

Certamente, a maioria dos projetos são mais complicados que nosso exemplo `calc`, contendo múltiplos subdiretórios. Os usuários do Subversion muitas vezes seguem um algoritmo específico ao usar ramos:

1. Copiar todo o “trunk” do projeto para um novo diretório de ramo.

2. Comutar somente *parte* do “trunk” da cópia de trabalho para espelhar o ramo.

Em outras palavras, se um usuário sabe que o trabalho no ramo só deve acontecer sobre um subdiretório específico, eles usam **svn switch** para mover somente este subdiretório para o ramo. (Ou algumas vezes os usuários comutarão apenas um único arquivo de trabalho para o ramo!) Dessa forma, eles podem continuar a receber normalmente as atualizações do “trunk” para a maior parte de sua cópia de trabalho, mas as porções comutadas ficarão imunes (a não ser que alguém submeta uma mudança em seu ramo). Esta funcionalidade adiciona uma completa nova dimensão ao conceito de uma “cópia de trabalho mista”—podemos ter não apenas cópias de trabalho que possuem uma mistura de revisões de trabalho, mas também uma mistura de locais de repositório.

Se sua cópia de trabalho contém um número de sub-árvores comutadas de diferentes locais do repositório, ela continua a funcionar normalmente. Quando você atualiza, você receberá as diferenças em cada sub-árvore apropriadamente. Quando você submete, suas mudanças locais ainda serão aplicadas como uma única e atômica mudança para o repositório.

Note que enquanto está tudo certo para sua cópia de trabalho refletir uma mistura de locais do repositório, estes locais devem estar todos dentro do *mesmo* repositório. Os repositórios do Subversion ainda não são capazes de comunicarem entre si; esta é uma funcionalidade planejada para o futuro. [4]

[4] Você *pode*, entretanto, usar **svn switch** com a opção `--relocate` se a URL de seu servidor mudar e você não quiser abandonar uma cópia de trabalho existente. Veja svn switch para mais informações e um exemplo.

Comutações e Atualizações

Você reparou que a saída dos comandos **svn switch** e **svn update** possuem a mesma aparência? O comando `switch` é na verdade um "super-comando" do comando `update`.

Quando você executa **svn update**, você está pedindo ao repositório para comparar duas árvores. O repositório assim faz, e então envia uma descrição das diferenças de volta para o cliente. A única diferença entre **svn switch** e **svn update** é que o comando `update` sempre compara dois caminhos idênticos.

Isto é, se sua cópia de trabalho é um espelho de `/calc/trunk`, então **svn update** comparará automaticamente sua cópia de trabalho de `/calc/trunk` com `/calc/trunk` na revisão `HEAD`. Se você está comutando sua cópia de trabalho para um ramo, então **svn switch** comparará sua cópia de trabalho de `/calc/trunk` com algum *outro* diretório de ramo na revisão `HEAD`.

Em outras palavras, uma atualização move sua cópia de trabalho através do tempo. Uma comutação move sua cópia de trabalho através do tempo *e* do espaço.

Porque **svn switch** é essencialmente uma variante de **svn update**, ele compartilha os mesmos comportamentos; qualquer modificação local em sua cópia de trabalho é preservada quando novos dados chegam do repositório. Isso lhe permite executar todos os tipos de truques engenhosos.

Por exemplo, suponha que você tem uma cópia de trabalho de `/calc/trunk` e realizou um certo número de mudanças nele. Então você rapidamente constata que você pretendia fazer as mudanças em um ramo. Não tem problema! Quando você executa **svn switch** para um ramo de sua cópia de trabalho, as mudanças locais permanecerão. Você pode então testar e submeter elas para o ramo.

Rótulos

Outro conceito comum do controle de versão é *ramo*. Um ramo é apenas uma "foto" do projeto no momento. No Subversion, essa idéia parece estar em todo lugar. Cada revisão do repositório é exatamente isso—uma foto da estrutura depois de cada commit.

Entretanto, pessoas normalmente querem dar rótulos mais amigáveis como nomes de tags, como `versão-1.0`. E querem fazer "fotos" de pequenos sub-diretórios da estrutura. Além do mais, não é fácil lembrar que versão-1.0 de um pedaço do software é um particular sub-diretório da revisão 4822.

Criando um rótulo simples

Mais uma vez, **snv copy** vem para nos socorrer. Se você quer criar uma foto do `/calc/trunk` exatamente como ele está na revisão `HEAD`, fazendo uma copia dela:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/tags/release-1.0 \
      -m "Rótulando a versão 1.0 do projeto 'calc'."

Committed revision 351.
```

Este exemplo assume que o diretório `/calc/tags` já existe. (Se ele não existir, você pode criá-lo usando **svn mkdir**.) Depois da copia completar, o novo diretório `versão-1.0` será para sempre uma foto de como o projeto estava na revisão `HEAD` no momento que a copia foi feita. Claro que você pode querer mais precisão em saber qual revisão a copia foi feita, em caso de alguém ter feito commit no projeto quando você não estava vendo. Então se você sabe que a revisão 350 do `/calc/trunk` é exatamente a foto que você quer, você pode especificar isso passando `-r 350` para o comando **svn copy**.

Mas espere um pouco: não é essa criação do rótulo o mesmo procedimento para criar um ramo? Sim, de fato, é. No Subversion, não há diferença entre um rótulo e um ramo. Assim como com ramos, a única razão uma cópia é um “rótulo” é porque *humanos* decidiram tratar isso desse jeito: desde que ninguém nunca faça commit para esse diretório, ele permanecerá para sempre uma foto. Se as pessoas começarem a fazer commit para ele, ele se transoforma num ramo.

Se você está administrando um repositório, existe duas maneiras para gerenciar rótulos. A primeira é “não toque”: como uma política do projeto, decida onde os rótulos vão morar, e garanta que todos os usuários saibam como tratar os diretórios que eles vão copiar para lá. (Isso quer dizer, garanta que eles saibam que não devem fazer neles.) A segunda é mais paranóica: você pode usar um dos scripts de controle de acesso providos com o Subversion para previnir que alguém faça algo além de apenas criar novas copias na área de rótulos (Veja Capítulo 6, *Configuração do Servidor*.) A maneira paranoica, entrentanto, não é necessária. Se algum usuário acidentalmente fizer commit de alguam mudança para o diretório de rótulo, você pode simplesmente desfazer a mudança como discutido na revisção anterior. É um controle de versão apesar de tudo.

Criando um rótulo complexo

Algumas vezes você que sua “foto” seja mais complicada que um simples diretório de uma única revisão.

Por exemplo, pense que seu projeto é muito maior que nosso exemplo `calc`: suponha que contém um número de sub-diretórios e muitos outros arquivos. No curso do seu trabalho, você pode decidir que você precisa criar um cópia de trabalho que é destinado para novos recursos e correções de erros. Você pode conseguir isso selecionando arquivos e diretórios com datas anteriores em uma revisão particular (usando **svn update -r** livremente), ou mudando arquivos e diretórios para um ramo em particular (fazendo uso do **svn switch**). Quando estiver pronto, sua cópia de trabalho será uma mistura de diferentes revisões. Mas depois de testes, você saberá que é exatamente a combinação que você precisa.

Hora de fazer a foto. Copiar uma URL para outra não vai funcionar aqui. Nesse caso, você quer fazer uma foto exata da cópia de trabalho que você organizou e armazenar no repositório. Felizmente, **svn copy** na verdade tem quatro diferentes maneiras de ser usado (você pode ler sobre em Capítulo 9, *Referência Completa do Subversion*), incluindo a habilidade de copiar uma árvore de cópia de trablho para o respositório:

```
$ ls
my-working-copy/

$ svn copy my-working-copy http://svn.example.com/repos/calc/tags/mytag

Committed revision 352.
```

Agora existe um novo diretório no respositório `/calc/tags/mytag`, que é uma foto exata da sua cópia de trabalho—combinado revisões, URLs, e tudo mais.

Outros usuários tem encontrado usos interessantes para esse recurso. Algumas vezes existe situações onde você tem um monte de mudanças locais na sua cópia de trabalho, e você gostaria que um colega de trabalho as visse. Ao invés de usar **svn diff** e enviar o arquivo patch (que não irá ter as informações de mudança na árvore de diretórios, em symlink e mudanças nas propriedades), você pode usar **svn copy** para “subir” sua cópia local para uma área privada no repositório. Seu colega pode verificar o nome de cópia da sua cópia de trabalho, ou usar **svn merge** para receber as exatas mudanças.

Sendo isso um método legal para subir uma rápida foto do seu trabalho local, note que isso *não* é uma boa maneira de iniciar um ramo. A criação de um ramo deve ser um evento solitário, e esse método exige a criação de um ramo com mudanças extras em arquivos, tudo em uma única revisão. Isso dificulta muito (mais tarde) a identificar um número de uma revisão como um ponto de um ramo.

Já se encontrou fazendo edições complexas (no sua cópia de trabalho `/trunk`) e de repente percebe, “Ei, estas mudanças deviam estar num ramo próprio?” Uma ótima técnica para fazer isso pode ser resumir em dois passos:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/branches/newbranch
Committed revision 353.

$ svn switch http://svn.example.com/repos/calc/branches/newbranch
At revision 353.
```

O comando **svn switch**, como **svn update**, preserva suas edições locais. Nesse ponto, sua cópia de trabablho é um reflexo do novo ramo criado, e seu próximo **svn commit** ira enviar suas mudanças para lá.

Manutenção de Ramos

Você pode ter notado por agora que o Subversion é extremamente flexível. Como ele implementa ramos e rótulos usando o mesmo mecanismo de suporte (cópias de diretório), e como ramos e rótulos aparecem normalmente no espaço do sistema de arquivos, muitas pessoas vêem o Subversion como algo intimidador. Ele é realmente *muito* flexível. Nesta seçãi, vamos lhe dar algumas sugestões sobre como arrajar e gerenciar seus dados ao longo do tempo.

Repository Layout

There are some standard, recommended ways to organize a repository. Most people create a `trunk` directory to hold the “main line” of development, a `branches` directory to contain branch copies, and a `tags` directory to contain tag copies. If a repository holds only one project, then often people create these top-level directories:

```
/trunk
/branches
/tags
```

If a repository contains multiple projects, admins typically index their layout by project (see “Planejando a Organização do Repositório” to read more about “project roots”):

```
/paint/trunk
/paint/branches
/paint/tags
/calc/trunk
/calc/branches
/calc/tags
```

Of course, you're free to ignore these common layouts. You can create any sort of variation, whatever works best for you or your team. Remember that whatever you choose, it's not a permanent commitment. You can reorganize your repository at any time. Because branches and tags are ordinary directories, the **svn move** command can move or rename them however you wish. Switching from one layout to another is just a matter of issuing a series of server-side moves; if you don't like the way things are organized in the repository, just juggle the directories around.

Remember, though, that while moving directories may be easy to do, you need to be considerate of your users as well. Your juggling can be disorienting to users with existing working copies. If a user has a working copy of a particular repository directory, your **svn move** operation might remove the path

from the latest revision. When the user next runs **svn update**, she will be told that her working copy represents a path that no longer exists, and the user will be forced to **svn switch** to the new location.

Data Lifetimes

Another nice feature of Subversion's model is that branches and tags can have finite lifetimes, just like any other versioned item. For example, suppose you eventually finish all your work on your personal branch of the `calc` project. After merging all of your changes back into `/calc/trunk`, there's no need for your private branch directory to stick around anymore:

```
$ svn delete http://svn.example.com/repos/calc/branches/my-calc-branch \
             -m "Removing obsolete branch of calc project."

Committed revision 375.
```

And now your branch is gone. Of course it's not really gone: the directory is simply missing from the `HEAD` revision, no longer distracting anyone. If you use **svn checkout**, **svn switch**, or **svn list** to examine an earlier revision, you'll still be able to see your old branch.

If browsing your deleted directory isn't enough, you can always bring it back. Resurrecting data is very easy in Subversion. If there's a deleted directory (or file) that you'd like to bring back into `HEAD`, simply use **svn copy -r** to copy it from the old revision:

```
$ svn copy -r 374 http://svn.example.com/repos/calc/branches/my-calc-branch \
                  http://svn.example.com/repos/calc/branches/my-calc-branch

Committed revision 376.
```

In our example, your personal branch had a relatively short lifetime: you may have created it to fix a bug or implement a new feature. When your task is done, so is the branch. In software development, though, it's also common to have two “main” branches running side-by-side for very long periods. For example, suppose it's time to release a stable version of the `calc` project to the public, and you know it's going to take a couple of months to shake bugs out of the software. You don't want people to add new features to the project, but you don't want to tell all developers to stop programming either. So instead, you create a “stable” branch of the software that won't change much:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
         http://svn.example.com/repos/calc/branches/stable-1.0 \
         -m "Creating stable branch of calc project."

Committed revision 377.
```

And now developers are free to continue adding cutting-edge (or experimental) features to `/calc/trunk`, and you can declare a project policy that only bug fixes are to be committed to `/calc/branches/stable-1.0`. That is, as people continue to work on the trunk, a human selectively ports bug fixes over to the stable branch. Even after the stable branch has shipped, you'll probably continue to maintain the branch for a long time—that is, as long as you continue to support that release for customers.

Ramos de fornecedores

Como é especialmente o caso quando se trata de desenvolvimento de software, os dados que você mantém sob controle de versão freqüentemente são intimamente relacionados a, ou talvez dependentes de, dados alheios. Geralmente, as necessidades do seu projeto determinarão que você fique tão atualizado quanto possível em relação aos dados fornecidos por essa entidade externa sem

sacrificar a estabilidade do seu próprio projeto. Este cenário se repete o tempo todo—em qualquer lugar onde a informação gerada por um grupo de pessoas tem um efeito direto sobre o que é gerado por outro grupo.

Por exemplo, desenvolvedores de software podem estar trabalhando em um aplicativo que faz uso de uma biblioteca de terceiros. O Subversion tem tal relacionamento com a biblioteca Apache Portable Runtime (ver "A Biblioteca Apache Portable Runtime"). O código fonte do Subversion depende da biblioteca APR para todas as suas necessidades de portabilidade. Em fases anteriores do desenvolvimento do Subversion, o projeto seguiu de perto as mudanças na API da APR, aderindo sempre ao "estado da arte" das agitações no código da biblioteca. Agora que tanto a APR quanto o Subversion amadureceram, o Subversion tenta sincronizar com a API da biblioteca APR somente em liberações estáveis e bem testadas.

Agora, se seu projeto depende de informações de alguém, existem várias maneiras pelas quais você poderia tentar sincronizar essas informações com as suas próprias. Mais dolorosamente, você poderia emitir oral instruções orais ou por escrito a todos os contribuintes do seu projeto, dizendo-lhes para se certificar de que têm as versões específicas dessa informação de terceiros de que seu projeto precisa. Se a informação de terceiros é mantida em um repositório Subversion, você também pode utilizar as definições externas do Subversion para efetivamente "imobilizar" versões específicas dessa informação em algum lugar no próprio diretório da sua cópia de trabalho (ver "Definições Externas").

Mas às vezes você quer manter modificações personalizadas de dados de terceiros em seu próprio sistema de controle de versões. Retornando ao exemplo do desenvolvimento de software, programadores poderiam precisar fazer modificações naquela biblioteca de terceiros para seus próprios propósitos. Estas alterações poderiam incluir novas funcionalidades ou correções de bugs, mantidas apenas internamente até se tornarem parte de uma liberação oficial da biblioteca de terceiros. Ou as mudanças poderiam nunca ser transmitidas de volta para os mantenedores da biblioteca, existindo apenas como ajustes personalizados para fazer com que a biblioteca atenda melhor as necessidades dos desenvolvedores de software.

Agora você enfrenta uma situação interessante. Seu projeto poderia abrigar suas próprias modificação aos dados de terceiros de algum modo desarticulado, tal como a utilização de *patches* ou versões alternativas completas dos arquivos e diretórios. Mas estas rapidamente tornam-se dores de cabeça para a manutenção, exigindo algum mecanismo para aplicar suas alterações personalizadas aos dados de terceiros, e necessitando de regeneração dessas mudanças a cada sucessiva versão dos dados de terceiros que você acompanha.

A solução para este problema é usar *ramos de fornecedores* (*vendor branches*). Um ramo de fornecedor é uma árvore de diretórios no nosso próprio sistema de controle de versões que contém informações fornecidas por uma entidade de terceiros, ou fornecedor. Cada versão dos dados do fornecedor que você decidir absorver em seu projeto é chamada *pingo de fornecedor* (*vendor drop*).

Os ramos de fornecedor proporcionam dois benefícios. Primeiro, por armazenar o pingo de fornecedor atualmente suportado em seu próprio sistema de controle de versão, os membros do seu projeto nunca precisam perguntar se têm a versão correta dos dados do fornecedor. Eles simplesmente recebem essa versão correta como parte de suas atualizações regulares da cópia de trabalho. Em segundo lugar, como os dados residem em seu próprio repositório Subversion, você pode armazenar as alterações personalizadas feitas nele no próprio local—você não precisa mais de um método automatizado (ou pior, manual) para incluir no projeto suas personalizações.

Procedimento Geral para Manutenção de Ramos de Fornecedores

Gerenciar ramos de fornecedores geralmente funciona assim. Você cria um diretório de nível superior (tal como `/vendor`) para conter os ramos de fornecedores. Então você importa o código de terceiros em um subdiretório desse diretório de nível superior. Em seguida copia esse subdiretório para o seu ramo principal de desenvolvimento (por exemplo, `/trunk`) no local apropriado. Você sempre faz suas alterações locais no ramo principal de desenvolvimento. A cada nova versão do código que você está

acompanhando, você o traz para o ramo de fornecedor e funde as alterações em `/trunk`, resolvendo quaisquer conflitos que ocorrerem entre suas alterações locais e as alterações da nova versão.

Talvez um exemplo ajudará a esclarecer este algoritmo. Usaremos um cenário onde a sua equipe de desenvolvimento está criando um *software* de calculadora que referencia uma complexa biblioteca de aritmética de terceiros, libcomplex. Começaremos com a criação inicial do ramo de fornecedor, e a importação do primeiro pingo de fornecedor. Iremos chamar o nosso diretório do ramo de fornecedor de `libcomplex`, e nossos pingos de código irão para um subdiretório do nosso ramo de fornecedor chamado `current`. E como **svn import** cria todos os diretórios pais intermediários de que precisa, nós podemos de fato realizar ambos os os passos com um único comando.

```
$ svn import /caminho/para/libcomplex-1.0 \
             http://svn.exemplo.com/repos/vendor/libcomplex/current \
             -m 'importando pingo de fornecedor 1.0 inicial'
…
```

Temos agora a versão atual do código fonte de libcomplex em `/vendor/libcomplex/current`. Agora, vamos rotular essa versão (ver "Rótulos") e então copiá-la para o ramo principal de desenvolvimento. Nosso cópia criará um novo diretório chamado `libcomplex` no nosso diretório de projeto `calc` já existente. É nesta versão copiada dos dados do fornecedor que nós vamos fazer nossas personalizações.

```
$ svn copy http://svn.exemplo.com/repos/vendor/libcomplex/current  \
           http://svn.exemplo.com/repos/vendor/libcomplex/1.0      \
           -m 'rotulando libcomplex-1.0'
…
$ svn copy http://svn.exemplo.com/repos/vendor/libcomplex/1.0  \
           http://svn.exemplo.com/repos/calc/libcomplex        \
           -m 'trazendo libcomplex-1.0 para o ramo principal'
…
```

Nós obtemos uma cópia do ramo principal do nosso projeto—que agora inclui uma cópia do primeiro pingo de fornecedor—e começamos a trabalhar personalizando o código de libcomplex. Antes que saibamos, nossa versão modificada de libcomplex agora está completamente integrada ao nosso programa da calculadora. [5]

Algumas semanas depois, os desenvolvedores da libcomplex lançam uma nova versão da sua biblioteca—versão 1.1—que contém algumas características e funcionalidades que nós queremos muito. Nós gostaríamos de atualizar para esta nova versão, mas sem perder as personalizações que fizemos na versão existente. O que nós essencialmente gostaríamos de fazer é substituir nossa atual versão base da libcomplex 1.0 por uma cópia da libcomplex 1.1 e, em seguida, voltar a aplicar as modificações que fizemos anteriormente na biblioteca, desta vez para a nova versão. Mas na prática nós abordamos o problema na outra direção, aplicando as alterações feitas em libcomplex entre as versões 1.0 e 1.1 diretamente na nossa cópia personalizada dela.

Para executar esta atualização, nós obtemos uma cópia do nosso ramo de fornecedor, e substituímos o código no diretório `current` pelo novo código fonte da libcomplex 1.1. Nós literalmente copiamos novos arquivos sobre os arquivos existentes, talvez descompactando a versão compactada da libcomplex 1.1 sobre nossos arquivos e diretórios existentes. A meta aqui é fazer nosso diretório `current` conter apenas o código da libcomplex 1.1, e garantir que todo esse código esteja sob controle de versão. Ah, e nós queremos fazer isto com o mínimo possível de perturbação no histórico do controle de versão.

Após substituir o código 1.0 pelo código 1.1, **svn status** vai mostrar arquivos com modificações locais assim como, talvez, alguns arquivos fora do controle de versão ou faltantes. Se nós fizemos o que deveríamos ter feito, os arquivos fora do controle de versão são apenas os novos arquivos introduzidos

[5] E inteiramente livre de *bugs*, é claro!

na versão 1.1 da libcomplex—nós executamos **svn add** sobre eles para colocá-los sob controle versão. Os arquivos faltantes são arquivos que estavam em 1.0, mas não em 1.1, e sobre esses caminhos nós executamos **svn delete**. Por fim, uma vez que nossa cópia de trabalho `current` contém apenas o código da libcomplex 1.1, nós submetemos as alterações que fizemos para que ela ficasse desse jeito.

Nosso ramo `current` agora contém o novo pingo de fornecedor. Nós rotulamos a nova versão (da mesma maneira que anteriormente rotulamos o pingo de fornecedor da versão 1.0), e em seguida fundimos as diferenças entre o rótulo da versão anterior e a nova versão atual em nosso ramo principal de desenvolvimento.

```
$ cd working-copies/calc
$ svn merge http://svn.exemplo.com/repos/vendor/libcomplex/1.0      \
            http://svn.exemplo.com/repos/vendor/libcomplex/current  \
            libcomplex
… # resolva todos os conflitos entre as alterações deles e as nossas
$ svn commit -m 'fundindo libcomplex-1.1 com o ramo principal'
…
```

No caso de uso trivial, a nova versão da nossa ferramenta de terceiros pareceria com a versão anterior, de um ponto de vista de arquivos e diretórios. Nenhum dos arquivos fonte de libcomplex teria sido excluído, renomeado ou movido para locais diferentes—a nova versão conteria apenas alterações textuais em relação à anterior. Em um mundo perfeito, nossas alterações seriam facilmente aplicadas à nova versão da biblioteca, sem absolutamente nenhuma complicação ou conflito.

Mas as coisas nem sempre são assim tão simples, e na verdade é bastante comum que arquivos fonte sejam movidos de lugar entre liberações de *software*. Isto dificulta o processo de garantir que as nossas alterações ainda são válidas para a nova versão do código, e pode degradar rapidamente em uma situação onde teremos de recriar manualmente as nossas customizações na nova versão. Uma vez que o Subversion conhece a história de um determinado arquivo fonte—incluindo todas as suas localizações anteriores—o processo de fusão da nova versão da biblioteca é bem simples. Mas nós somos responsáveis por dizer ao Subversion como a posição do arquivo fonte mudou entre um pingo de fornecedor e outro.

svn_load_dirs.pl

Vendor drops that contain more than a few deletes, additions and moves complicate the process of upgrading to each successive version of the third-party data. So Subversion supplies the **svn_load_dirs.pl** script to assist with this process. This script automates the importing steps we mentioned in the general vendor branch management procedure to make sure that mistakes are minimized. You will still be responsible for using the merge commands to merge the new versions of the third-party data into your main development branch, but **svn_load_dirs.pl** can help you more quickly and easily arrive at that stage.

In short, **svn_load_dirs.pl** is an enhancement to **svn import** that has several important characteristics:

- It can be run at any point in time to bring an existing directory in the repository to exactly match an external directory, performing all the necessary adds and deletes, and optionally performing moves, too.
- It takes care of complicated series of operations between which Subversion requires an intermediate commit—such as before renaming a file or directory twice.
- It will optionally tag the newly imported directory.
- It will optionally add arbitrary properties to files and directories that match a regular expression.

svn_load_dirs.pl takes three mandatory arguments. The first argument is the URL to the base Subversion directory to work in. This argument is followed by the URL—relative to the first argument—

into which the current vendor drop will be imported. Finally, the third argument is the local directory to import. Using our previous example, a typical run of **svn_load_dirs.pl** might look like:

```
$ svn_load_dirs.pl http://svn.example.com/repos/vendor/libcomplex \
                   current                                        \
                   /path/to/libcomplex-1.1
…
```

You can indicate that you'd like **svn_load_dirs.pl** to tag the new vendor drop by passing the `-t` command-line option and specifying a tag name. This tag is another URL relative to the first program argument.

```
$ svn_load_dirs.pl -t libcomplex-1.1                              \
                   http://svn.example.com/repos/vendor/libcomplex \
                   current                                        \
                   /path/to/libcomplex-1.1
…
```

When you run **svn_load_dirs.pl**, it examines the contents of your existing “current” vendor drop, and compares them with the proposed new vendor drop. In the trivial case, there will be no files that are in one version and not the other, and the script will perform the new import without incident. If, however, there are discrepancies in the file layouts between versions, **svn_load_dirs.pl** will ask you how to resolve those differences. For example, you will have the opportunity to tell the script that you know that the file `math.c` in version 1.0 of libcomplex was renamed to `arithmetic.c` in libcomplex 1.1. Any discrepancies not explained by moves are treated as regular additions and deletions.

The script also accepts a separate configuration file for setting properties on files and directories matching a regular expression that are *added* to the repository. This configuration file is specified to **svn_load_dirs.pl** using the `-p` command-line option. Each line of the configuration file is a whitespace-delimited set of two or four values: a Perl-style regular expression to match the added path against, a control keyword (either `break` or `cont`), and then optionally a property name and value.

```
\.png$              break   svn:mime-type   image/png
\.jpe?g$            break   svn:mime-type   image/jpeg
\.m3u$              cont    svn:mime-type   audio/x-mpegurl
\.m3u$              break   svn:eol-style   LF
.*                  break   svn:eol-style   native
```

For each added path, the configured property changes whose regular expression matches the path are applied in order, unless the control specification is `break` (which means that no more property changes should be applied to that path). If the control specification is `cont`—an abbreviation for `continue`—then matching will continue with the next line of the configuration file.

Any whitespace in the regular expression, property name, or property value must be surrounded by either single or double quote characters. You can escape quote characters that are not used for wrapping whitespace by preceding them with a backslash (`\`) character. The backslash escapes only quotes when parsing the configuration file, so do not protect any other characters beyond what is necessary for the regular expression.

Sumário

Nós cobrimos muito chão nesse capítulo. Nós discutimos conceitos de rótulo*tags* e ramos *branches*, e demonstramos como Subversion implementa estes conceitos através da cópia de diretórios com o comando **svn copy**. Nós mostramos como usar **svn merge** para copiar mudanças de um ramo *branch* para outro, ou reverter mudanças indesejadas. Nós passamos pelo uso do **svn switch** para criar locais

mistos de copias de trabalho. E nós falamos sobre como eles podem gerenciar a organização e vida dos ramos *branches* em um respositório.

Lembre-se do mantra do Subversion: ramos *branches* e rótulos *tags* são baratos. Então use-os livremente! Ao mesmo tempo, não esqueça de usar bons hábitos de fusão *merge*. Cópias baratas são úteis apenas quando você é cuidadoso ao rastrear suas fusões *merges*.

Capítulo 5. Administração do Repositório

O repositório Subversion é a central de todos os dados que estão sendo versionados. Assim, ele se transforma num candidato óbvio para receber todo amor e atenção que um administrador pode oferecer. Embora o repositório seja geralmente um item de baixa manutenção, é importante entender como configurar e cuidar apropriadamente para que problemas potenciais sejam evitados, e problemas eventuais sejam resolvidos de maneira segura.

Neste capítulo, vamos discutir sobre como criar e configurar um repositório Subversion. Vamos falar também sobre manutenção, dando exemplos de como e quando usar as ferramentas **svnlook** e **svnadmin** providas pelo Subversion. Vamos apontar alguns questionamentos e erros, e dar algumas sugestões sobre como organizar seus dados em um repositório.

Se você planeja acessar um repositório Subversion apenas como um usuário cujos dados estão sendo versionados (isto é, por meio de um cliente Subversion), você pode pular esse capítulo todo. Entretanto, se você é, ou deseja se tornar, um administrador de um repositório Subversion, [1] este capítulo é para você.

O Repositório Subversion, Definição

Antes de entrarmos no vasto tópico da administração do repositório, vamos primeiro definir o que é um repositório. Como ele se parece? Como ele se sente? Ele gosta de chá gelado ou quente, doce, e com limão? Como um administrador, será esperado que você entenda a composição de um repositório tanto da perspectiva do Sistema Operacional—como o repositório se parece e se comporta em relação a ferramentas que não são do Subversion—e de uma perspectiva lógica—relacionada com a forma com que os dados são representados *dentro* do repositório.

Vendo pelos olhos de um típico navegador de arquivos (como o Windows Explorer) ou de ferramentas de navegação em sistemas de arquivos baseadas em linha de comando, o repositório Subversion é apenas outro diretório cheio de coisas. Existem alguns subdiretórios que possuem arquivos de configuração que podem ser lidos por humanos, e outros que não são tão fáceis de serem lidos, e assim por diante. Como em outras áreas do projeto do Subversion, modularidade tem grande importância, e a organização hierárquica é usada pra controlar o caos. Assim, uma olhada superficial nas partes essenciais é suficiente para revelar os componentes básicos do repositório:

```
$ ls repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

Aqui está uma pequena pincelada do que exatamente você está vendo nessa lista do diretório. (Não fique assustado com a terminologia—uma explicação mais detalhada desses componentes está disponível em algum lugar nesse e em outros capítulos.)

conf
: Um diretório contendo arquivos de configuração do repositório.

dav
: Um diretório onde ficam os arquivos usados pelo mod_dav_svn.

db
: Local onde são armazenados todos os seus dados versionados.

[1] Isto pode soar bem metido ou arrogante, mas nós estamos apenas falando de alguém que tenha interesse no misterioso local por trás das cópias de trabalho onde os dados de todos ficam.

format
: Um arquivo que contém um simples inteiro que indica o número da versão do repositório.

hooks
: Um diretório cheio de modelos de scripts (e scripts, uma vez que você tenha instalado algum).

locks
: Um diretório para arquivos travados do Subversion, usado para rastrear acessos ao repositório.

README.txt
: Arquivo que meramente informa a seus leitores que eles estão olhando para um repositório Subversion.

É claro que, quando acessado por meio das bibliotecas do Subversion, esse estranho conjunto de arquivos e diretórios de repente torna-se uma implementação de um sistema de arquivos virtual, versionável e completo, com gatilhos de eventos personalizáveis. Este sistema de arquivos tem o seu próprio entendimento sobre diretórios e arquivos, muito semelhante aos conceitos usados em sistemas de arquivos reais (como NTFS, FAT32, ext3, e assim por diante). Mas este é um sistema de arquivos especial—ele controla os diretórios e arquivos a partir das revisões, mantendo todas as mudanças que você fez neles armazenadas com segurança e sempre acessíveis. É aqui onde todos os seus dados versionados vivem.

Estratégias para Implementação de Repositórios

Devido, em grande parte, a simplicidade do projeto do repositório Subversion e as tecnologias nas quais ele se baseia, criá-lo e configurá-lo são tarefas bastante naturais. Existem algumas decisões preliminares que você precisará tomar, mas o trabalho necessário para fazer alguma configuração no repositório Subversion é muito simples, tendendo a repetição mecânica a medida que você começa a configurar várias dessas coisas.

Algumas coisas que você precisará considerar logo no início são:

- Que dados você espera armazenar no seu repositório (ou repositórios), e como eles serão organizados?
- Onde viverá o seu repositório, e como ele será acessado?
- Que tipo de controle de acesso e notificação de eventos você irá precisar?
- Qual tipo de armazenamento de dados, entre os disponíveis, você irá utilizar?

Nessa seção nós iremos tentar ajudá-lo a responder essas questões.

Planejando a Organização do Repositório

Embora o Subversion permita que você mova arquivos e diretórios versionados sem qualquer perda de informação, e até mesmo provê meios de mover conjuntos inteiros de eventos históricos versionados de um repositório para outro, fazer isso pode atrapalhar significativamente o fluxo de trabalho daqueles que acessam o repositório frequentemente e esperam que algumas coisas estejam em certos lugares. Então antes de criar um novo repositório, tente olhar um pouco para o futuro; pense a diante antes de colocar seus dados no controle de versão. Planejando conscientemente o “leiaute” do repositório, ou repositórios, e seu conteúdo versionado antes do tempo, você pode prevenir muitas dores de cabeça futuras.

Vamos assumir que como administrador de repositório você será responsável pelo suporte do sistema de controle de versões para vários projetos. Sua primeira decisão é se usará um único repositório para

múltiplos projetos, ou fornecer para cada projeto o seu próprio repositório, ou ainda alguma combinação disso.

Existem vantagens em se utilizar um único repositório para múltiplos projetos e a mais óbvia é a ausência de manutenção duplicada. Um único repositório significa que haverá um único conjunto de programas de gatilhos, uma única coisa para fazer cópias de segurança periódicas, uma única coisa para descarregar e carregar se o Subversion lança um nova versão incompatível, e por aí vai. Além disso, você pode mover dados entre projetos facilmente, e sem perder qualquer informação de versionamento.

A desvantagem de usar um único repositório é que diferentes projetos podem ter diferentes requisitos em termos de gatilhos de eventos, como por exemplo a necessidade de enviar notificações de submissão para diferentes listas de e-mail, ou ter diferentes definições sobre o que constitui uma submissão correta. É claro que eles não são problemas insuperáveis—somente significa que todos os seus scripts de gatilho devem ser sensíveis ao leiaute do seu repositório ao invés de assumir que todo o repositório está associado com um único grupo de pessoas. Além disso, lembre-se que o Subversion usa números de revisão globais com relação ao repositório. Muito embora esses números não tenham particularmente nenhum poder mágico, algumas pessoas continuam não gostando do fato de que mesmo que não hajam modificações no seu projeto recentemente o número de revisão continua sendo incrementado porque outros projetos continuam adicionando novas revisões. [2]

Uma abordagem meio termo pode ser utilizada também. Por exemplo, projetos podem ser agrupados pela forma como eles se relacionam entre si. Você pode ter alguns poucos repositórios com um punhado de projetos em cada um deles. Dessa forma, projetos nos quais é desejável o compartilhamento de dados podem fazê-lo facilmente, e quando novas revisões são adicionadas ao repositório, os desenvolvedores saberão que essas revisões são no mínimo remotamente relacionadas com todos que usam esse repositório.

Depois de decidir como organizar seus projetos com relação aos repositórios você irá provavelmente pensar sobre a hierarquia de diretórios lá dentro. Como o Subversion utiliza cópias comuns de diretórios para ramificações (*branches*) e rótulos (*tags*) (veja Capítulo 4, *Fundir e Ramificar*), a comunidade recomenda que você escolha uma localização para cada *raiz de projeto*—o “mais alto” diretório que irá conter dados relacionados com o projeto—e então criar três subdiretórios abaixo desse raiz: `trunk`, o diretório no qual o desenvolvimento principal do projeto ocorre; `branches`, diretório no podem ser criados vários ramos da linha principal de desenvolvimento; `tags`, diretório que poderá conter uma coleção de instantâneos de árvores de diretório que são criados, e possivelmente destruídos, mas nunca alterados. [3]

Por exemplo, seu repositório poderá se parecer com o seguinte:

```
/
   calc/
      trunk/
      tags/
      branches/
   calendar/
      trunk/
      tags/
      branches/
   spreadsheet/
      trunk/
      tags/
      branches/
   …
```

[2]Quer seja baseado na ignorância ou em fracos conceitos sobre como produzir métricas de desenvolvimento corretamente, números de revisões globais são uma coisa tola para temer, e *não* o tipo de coisa que você deveria pesar na hora de decidir como organizar seus projetos e repositórios.

[3]O trio `trunk`, `tags`, e `branches` são muitas vezes chamados de “diretórios TTB”.

Note que não importa onde está cada raiz de projeto no seu repositório. Se você possuir somente um único projeto por repositório, o lugar mais lógico para colocar cada raiz de projeto é na raiz do respectivo repositório do projeto. Se você possui múltiplos projetos, você pode querer organizá-los em grupos dentro do repositório, talvez colocando projetos com objetivos semelhantes ou código compartilhado no mesmo subdiretório, ou talvez simplesmente agrupá-los alfabeticamente. Tal organização poderia se parecer com o que segue:

```
/
   utils/
      calc/
         trunk/
         tags/
         branches/
      calendar/
         trunk/
         tags/
         branches/
      …
   office/
      spreadsheet/
         trunk/
         tags/
         branches/
      …
```

Organize seu repositório da forma que você preferir. O Subversion não espera ou força uma organização particular—na sua visão, um diretório é um diretório. No final das contas você deve escolher a organização de repositório que atende as necessidades das pessoas que trabalham nos projetos que irão viver lá.

Em nome da revelação completa, no entanto, nós iremos mencionar outra forma muito comum de organização. Nesse leiaute os diretórios `trunk`, `tags` e `branches` residem no diretório raiz do repositório e os projetos estão em subdiretórios abaixo deles, como:

```
/
   trunk/
      calc/
      calendar/
      spreadsheet/
      …
   tags/
      calc/
      calendar/
      spreadsheet/
      …
   branches/
      calc/
      calendar/
      spreadsheet/
      …
```

Não existe nada de incorreto nessa forma de organização, mas ela pode ou não parecer intuitiva para seus usuários. Especialmente em situações de vários e grandes projetos com muitos usuários, esses usuários podem tender a se familiarizar com somente um ou dois projetos no repositório. Utilizar projetos como ramos irmãos tende a desenfatizar a individualidade dos projetos e focar no conjunto inteiro como uma única entidade. De qualquer forma essa é uma questão social. Nós gostamos da organização inicialmente proposta por razões puramente práticas—é fácil perguntar sobre (ou

modificar, ou migrar para outro lugar) o histórico completo de um único projeto quando existe um único caminho no repositório que guarda tudo—passado, presente, etiquetas, e ramos—referente ao projeto sozinho.

Decidindo Onde e Como Hospedar Seu Repositório

Antes de criar seu repositório Subversion, uma questão óbvia que você precisa responder é onde a coisa toda deverá ficar. Isto está fortemente interligado a uma miríade de outras questões que dizem respeito a como o repositório será acessado (através de um servidor Subversion ou diretamente), por quem (usuários que estejam por atrás de um firewall corporativo ou globalmente por meio da Internet), que outros serviços você irá disponibilizar em conjunto com o Subversion (interfaces de navegação de repositório, avisos de submissões (*commits*) por e-mail, etc.), sua estratégia de cópias de segurança (*backup*), e por aí vai.

Abordamos a escolha do servidor e configuração em Capítulo 6, *Configuração do Servidor*, o que gostaríamos brevemente de apontar aqui é simplesmente que as respostas a algumas destas e de outras perguntas podem ter implicações que forcem sua cuca ao decidir sobre onde seu repositório irá residir. Por exemplo, certos cenários de produção podem demandar acesso ao repositório a partir de um sistema de arquivos remoto para múltiplos computadores, caso este em que (como você verá na próxima seção) a sua escolha de um repositório secundário para armazenamento de dados passa a não ser uma escolha porque apenas um dos servidores secundários disponíveis irá funcionar neste cenário.

Averiguar cada possível maneira de implantação do Subversion também é impossível, e fora do escopo deste livro. Simplesmente encorajamos você a avaliar suas opções usando estas páginas e outras fontes como seu material de referência, e planejar a partir daí.

Escolhendo uma Base de Dados

A partir da versão 1.1, o Subversion oferece duas opções de tipos de base de dados—frequentemente referenciada como o “back-end” ou, de uma maneira que causa confusão, “o sistema de arquivos (versionado)”—que poderão ser utilizadas pelos repositórios. Um tipo mantém tudo em um ambiente de banco de dados Berkeley DB (ou BDB); repositórios baseados nesse ambiente também são conhecidos como “BDB-backed”. O outro tipo de armazenagem de dados usa arquivos comuns, com um formato próprio. Os desenvolvedores do Subversion têm o hábito de chamar esse último mecanismo de *FSFS* [4]—uma implementação de sistema de arquivos versionado que usa diretamente o sistema de arquivos nativo do Sistema Operacional—ao invés de uma biblioteca de banco de dados ou outra camada de abstração— para armazenar os dados.

Tabela 5.1, “Comparativo dos Mecanismos de Armazenamento” fornece um comparativo geral dos repositórios Berkeley DB e FSFS.

Tabela 5.1. Comparativo dos Mecanismos de Armazenamento

Categoria	Característica	Berkeley DB	FSFS
Confiabilidade	Integridade dos Dados	quando corretamente implementado é extremamente confiável; Berkeley DB 4.4 oferece auto-recuperação	versões antigas têm bugs que comprometem os dados, mas essas situações acontecem raramente
	Sensibilidade a interrupções	grande; travamentos e problemas de permissões podem deixar a base de dados	bastante insensível

[4] Frequentemente pronunciado como “fuzz-fuzz”, a menos que Jack Repenning tenha algo a dizer sobre isso. (Este livro, entretanto, assume que o leitor está pensando “efe-esse-efe-esse”.)

Categoria	Característica	Berkeley DB	FSFS
		“quebrada”, requirindo procedimentos para recuperação.	
Acessibilidade	Usável de uma sistema de arquivos "montado" como "somente leitura"	não	sim
	Armazenamento independente de plataforma	não	sim
	Usável em sistemas de arquivos de rede	geralmente não	sim
	Tratamento de permissões em grupo	sensível a problemas de umask; melhor se acessado por somente um usuário	contorna problemas de umask
Escalabilidade	Uso de disco do repositório	grande (especialmente se arquivos de log não são limpados)	pequeno
	Número de árvores de revisão	banco de dados; sem problemas	alguns sistemas de arquivos nativos antigos não crescem bem com milhares de entradas em um único diretório
	Diretórios com muitos arquivos	lento	rápido
Desempenho	Obter cópia da última revisão	sem diferenças significativas	sem diferenças significativas
	Grandes submissões	geralmente lentas, mas o custo é pago ao longo da vida da submissão	geralmente rápidas, mas atraso na finalização pode ocasionar *timeouts* no cliente

Existem vantagens e desvantagens em cada um desses dois mecanismos de armazenamento. Nenhum deles é mais “oficial” que o outro, embora os novos FSFS sejam o padrão do Subversion 1.2. Ambos são seguros o bastante para você confiar seus dados versionados. Mas como você pode ver em Tabela 5.1, “Comparativo dos Mecanismos de Armazenamento”, o FSFS oferece um pouco mais de flexibilidade em termos de seus cenários de implantação. Maior flexibilidade significa que você tem que trabalhar um pouco mais para encontrar formas de implantá-lo incorretamente. Essas razões—adicionadas ao fato de que não usando Berkeley DB significa que existem um componente a menos no sistema—explicam porque atualmente quase todo mundo utiliza o FSFS para criar novos repositórios.

Felizmente, muitos programas que acessam os repositórios Subversion são abençoados por ignorarem o mecanismo de armazenamento que está em uso. E você nem mesmo precisa se preocupar com a sua primeira escolha de mecanismo de armazenamento—no caso de você mudar de idéia posteriormente, o Subversion oferece formas de migrar os dados do seu repositório para outro repositório que usa um mecanismo de armazenamento diferente. Nós iremos falar mais sobre isso nesse capítulo.

As seguintes subseções oferecem uma visão mais detalhada sobre os tipos de mecanismos de armazenamento disponíveis.

Berkeley DB

Quando a fase de projeto inicial do Subversion estava em andamento, os desenvolvedores decidiram usar o Berkeley DB por diversas razões, incluindo sua licença open-source, suporte a transações,

confiabilidade, desempenho, simplicidade da API, segurança no uso em multitarefas, suporte para cursores de dados, dentre outras.

O Berkeley DB oferece suporte real para transações—talvez seu recurso mais poderoso. Múltiplos processos que acessem seus repositórios Subversion não precisam se preocupar em sobrescrever acidentalmente os dados uns dos outros. O isolamento oferecido pelo sistema de transações age de tal forma que, para cada operação realizada, o código no repositório do Subversion tenha uma visão estática da base de dados—e não uma base de dados que esteja mudando constantemente nas mãos de alguns outros processos—e possa tomar decisões baseadas no que vê. Se uma decisão tomada parecer conflitar com o que outro processo esteja fazendo, a operação inteira é desfeita como se nunca tivesse acontecido, e o Subversion graciosamente irá tentar executar a operação sobre uma nova, atualizada (e ainda assim, estática) visão da base de dados.

Outro grande recurso do Berkeley DB é o *backup a quente*—a habilidade de executar uma cópia de segurança do ambiente da base de dados sem precisar deixar o sistema “offline”. Vamos discutir como fazer cópias de segurança de seu repositório em “Repository Backup”, mas os benefícios de se poder fazer cópias funcionais de seus repositórios sem desligar o sistema devem lhe ser óbvios.

Berkeley DB é também um sistema de base de dados muito confiável quando utilizado adequadamente. O Subversion usa as facilidades de registros de log do Berkeley DB, o que quer dizer que a base de dados primeiro escreve uma descrição de quaisquer modificações que estiver para fazer em seus arquivos de log em disco, e só então realiza a modificação em si. Isto é para garantir que se qualquer coisa der errado, o sistema da base de dados pode se recuperar para um determinado *ponto de verificação* (*checkpoint*) anterior—um local nos arquivos de log que se sabe não estarem corrompidos —e re-executa as transações até que os dados sejam restaurados para um estado utilizável. Veja “Managing Disk Space” para saber mais sobre os arquivos de log do Berkeley DB.

Mas cada rosa tem seus espinhos, e assim devemos destacar algumas conhecidas limitações do Berkeley DB. Em primeiro lugar, o ambiente do Berkeley DB não é portável. Você não pode simplesmente copiar um repositório do Subversion que foi criado em um sistema Unix para dentro de um sistema Windows e esperar que funcione. Ainda que muito do formato da base de dados do Berkeley DB seja independente de plataforma, há alguns aspectos do ambiente que não o são. Em segundo lugar, o Subversion utiliza o Berkeley DB de forma que não irá funcionar em sistemas Windows 95/98—se você precisa hospedar um repositório em formato Berkeley DB em uma máquina Windows, utilize-o com sistemas Windows 2000 ou posteriores.

Ainda que o Berkeley DB prometa se comportar corretamente em compartilhamentos de rede que estejam de acordo com um conjunto de especificações, [5] e a maioria dos tipos de sistemas de arquivos e aplicações atuais *não* implementam essas tais especificações. E de forma nenhuma você pode usar um repositório baseado em BDB que resida em um compartilhamento de rede sendo acessado por múltiplos clientes do compartilhamento de uma só vez (o que muitas vezes é o ponto determinante para não se escolher ter repositórios hospedados em um compartilhamento em primeiro lugar).

Se você tentar usar Berkeley DB em um sistema de arquivos remoto que não atenda às especificações, os resultados serão imprevisíveis—você pode ver erros misteriosos imediatamente, ou pode levar meses antes de você descobrir que sua base de dados do repositório está sutilmente corrompida. Você deveria considerar muito seriamente o uso de armazenamento de dados em FSFS para repositórios que precisem residir em um compartilhamento de rede.

E finalmente, como o Berkeley DB é uma biblioteca interligada diretamente dentro do Subversion, ele é mais sensível a interrupções do que um sistema de uma base de dados relacional. A maioria dos sistemas SQL, por exemplo, possuem um processo servidor dedicado que intermedia todo o acesso às tabelas. Se um programa que esteja acessando a base de dados travar por algum motivo, o daemon da base de dados percebe a perda de conexão e efetua uma limpeza de quaisquer vestígios problemáticos que tenham ficado. E como o daemon da base de dados é o único processo que

[5]O Berkeley DB precisa que o sistema de arquivos em questão onde esteja o compartilhamento deve implementar estritamente a semântica POSIX de travamento, e ainda mais importante, a capacidade de mapear arquivos diretamente para processos em memória.

efetivamente acessa as tabelas, as aplicações não precisam se preocupar com relação a conflitos de permissão. No entanto, esse tipo de cenário não se aplica ao Berkeley BD. O Subversion (e os programas que usam bibliotecas do Subversion) acessam as tabelas da base de dados diretamente, o que quer dizer que o travamento de um programa pode deixar a base de dados temporariamente inconsistente, em um estado inacessível. Quando isso acontece, um administrador precisa solicitar que o Berkeley DB restaure-se a partir de um ponto de verificação, o que um tanto quanto inconveniente. Outras coisas também podem “comprometer” um repositório em virtude de processos travados, tais como conflitos entre programas relacionados a permissões e propriedades dos arquivos na base de dados.

O Berkeley DB 4.4 oferece para o Subversion (nas versões do Subversion 1.4 e superiores) a capacidade de se recuperar o ambiente do Berkeley DB de forma automática e transparente, caso necessário. Quando um processo do Subversion se interliga a um ambiente Berkeley DB do repositório, ele utiliza alguns mecanismos de contabilização para detectar quaisquer desconexões de processos anteriores, executa alguma recuperação necessária, e então prossegue como se nada tiver acontecido. Isto não elimina completamente as possibilidade de corrupção de instâncias do repositório, mas reduz drasticamente a necessidade de interação humana necessária para se recuperar desses tipos de problemas.

Assim, por mais que que um repositório Berkeley DB seja bastante rápido e escalável, ele é melhor aproveitado se for utilizado por um único processo servidor executando como um único usuário—como o **httpd** do Apache ou o **svnserve** (veja Capítulo 6, *Configuração do Servidor*)—ao invés do que por vários usuários diferentes através de URLs `file://` ou `svn+ssh://`. Se você for usar um repositório em Berkeley DB diretamente para múltiplos usuários, certifique-se de ler “Dando Suporte a Múltiplos Métodos de Acesso ao Repositório”.

FSFS

Em meados de 2004, um segundo tipo de sistema de armazenamento de repositórios—um que não usa um banco de dados— nasceu. Um repositório FSFS armazena as mudanças relacionadas com uma revisão em um única arquivo, e assim todas as revisões do repositório podem ser encontradas num único subdiretório que contém arquivos numerados. Transações são criadas em subdiretórios diferentes como arquivos individuais. Quando completa, o arquivo de transação é renomeado e movido para o diretório de revisões, garantindo assim que a transação será atômica. Em função do arquivo de revisão ser permanente e imodificável, também é possível fazer uma cópia de segurança “quente” assim como os repositórios baseados em BDB.

Os arquivos de revisão FSFS decrevem uma estrutura de diretórios de uma revisão, conteúdo dos arquivos, e diferenças em relação aos arquivos em outras árvores de revisão. Ao contrário das baseas de dados Berkeley DB esse formato de armazenamento é portável em muitos sistemas operacionais e não é sensível a arquitetura de CPU. Em função de não haver journaling ou arquivos com memória compartilhada o repositório pode ser acessado com segurança a partir de um sistema de arquivos de rede e examinado em um ambiente somente para leitura. A inexistência da sobrecarga de um banco de dados também significa que o tamanho total do repositório também é um pouco menor.

FSFS tem característica de desempenho diferentes também. Quando se submete um diretório com um alto número de arquivos, o FSFS é capaz de rapidamente atualizar as entradas de diretório. Por outro lado, FSF escreve a última versão de um arquivo como um delta em relação à uma versão anterior, o que significa que obtee a última árvore de diretórios é um pouco mais lento do que obter os textos completos armazenados em uma revisão HEAD no Berkeley DB. O FSFS também possui uma certa demora na finalização de uma submissão, o que em casos extremos pode causar timeout no cliente.

A diferença mais importante, entretanto, é o formato “inabalável” do FSFS quando alguma coisa errada acontece. Se ocorre um problema qualquer com um processo que está usando um banco de dados Berkeley DB, o banco de dados pode ficar em um estado que não permite o seu uso até que um adminitrador recupere ele. Se os mesmo cenários acontecerem com um processo que utiliza FSFS, o repositório não é afetado. No pior caso, algumas informações de transação são deixadas para trás.

O único argumento coerente contra o FSFS é que ele é relativamente imaturo quando comparado ao Berkeley DB. Ao contrário do Berkeley DB que tem anos de história, sua própria equipe de desenvolvimento e, agora, o grande nome da Oracle ligado à ele, [6] FSFS é muito mais novo em termos de engenharia. Antes da versão 1.4 ele ainda era afetado por algumas falhas bem sérias com relação a integridade dos dados, muito embora essas falhas ocorressem raramente, elas nunca deveriam ocorrer. Dito isso, o FSFS tem se tornado rapidamente a escolha de armazenamento de alguns dos maiores repositórios Subversion públicos e privados, e oferece poucos obstáculos a ponto de ser um bom ponto de entrada para o Subversion.

Creating and Configuring Your Repository

In “Estratégias para Implementação de Repositórios”, we looked at some of the important decisions that should be made before creating and configuring your Subversion repository. Now, we finally get to get our hands dirty! In this section, we'll see how to actually create a Subversion repository and configure it to perform custom actions when special repository events occur.

Creating the Repository

Subversion repository creation is an incredibly simple task. The **svnadmin** utility that comes with Subversion provides a subcommand (`create`) for doing just that.

```
$ svnadmin create /path/to/repos
```

This creates a new repository in the directory `/path/to/repos`, and with the default filesystem data store. Prior to Subversion 1.2, the default was to use Berkeley DB; the default is now FSFS. You can explicitly choose the filesystem type using the `--fs-type` argument, which accepts as a parameter either `fsfs` or `bdb`.

```
$ # Create an FSFS-backed repository
$ svnadmin create --fs-type fsfs /path/to/repos
$
```

```
# Create a Berkeley-DB-backed repository
$ svnadmin create --fs-type bdb /path/to/repos
$
```

After running this simple command, you have a Subversion repository.

The path argument to **svnadmin** is just a regular filesystem path and not a URL like the **svn** client program uses when referring to repositories. Both **svnadmin** and **svnlook** are considered server-side utilities—they are used on the machine where the repository resides to examine or modify aspects of the repository, and are in fact unable to perform tasks across a network. A common mistake made by Subversion newcomers is trying to pass URLs (even “local” `file://` ones) to these two programs.

Present in the `db/` subdirectory of your repository is the implementation of the versioned filesystem. Your new repository's versioned filesystem begins life at revision 0, which is defined to consist of nothing but the top-level root (`/`) directory. Initially, revision 0 also has a single revision property, `svn:date`, set to the time at which the repository was created.

Now that you have a repository, it's time to customize it.

While some parts of a Subversion repository—such as the configuration files and hook scripts—are meant to be examined and modified manually, you shouldn't (and shouldn't need to) tamper with the other parts of the repository “by hand”. The **svnadmin** tool should

[6] A Oracle comprou Sleepycat e seu software, Berkeley DB, no dia dos namorados em 2006.

be sufficient for any changes necessary to your repository, or you can look to third-party tools (such as Berkeley DB's tool suite) for tweaking relevant subsections of the repository. Do *not* attempt manual manipulation of your version control history by poking and prodding around in your repository's data store files!

Implementing Repository Hooks

A *hook* is a program triggered by some repository event, such as the creation of a new revision or the modification of an unversioned property. Some hooks (the so-called “pre hooks”) run in advance of a repository operation and provide a means by which to both report what is about to happen and to prevent it from happening at all. Other hooks (the “post hooks”) run after the completion of a repository event, and are useful for performing tasks that examine—but don't modify—the repository. Each hook is handed enough information to tell what that event is (or was), the specific repository changes proposed (or completed), and the username of the person who triggered the event.

The `hooks` subdirectory is, by default, filled with templates for various repository hooks.

```
$ ls repos/hooks/
post-commit.tmpl      post-unlock.tmpl  pre-revprop-change.tmpl
post-lock.tmpl        pre-commit.tmpl   pre-unlock.tmpl
post-revprop-change.tmpl  pre-lock.tmpl     start-commit.tmpl
```

There is one template for each hook that the Subversion repository supports, and by examining the contents of those template scripts, you can see what triggers each script to run and what data is passed to that script. Also present in many of these templates are examples of how one might use that script, in conjunction with other Subversion-supplied programs, to perform common useful tasks. To actually install a working hook, you need only place some executable program or script into the `repos/hooks` directory which can be executed as the name (like **start-commit** or **post-commit**) of the hook.

On Unix platforms, this means supplying a script or program (which could be a shell script, a Python program, a compiled C binary, or any number of other things) named exactly like the name of the hook. Of course, the template files are present for more than just informational purposes—the easiest way to install a hook on Unix platforms is to simply copy the appropriate template file to a new file that lacks the `.tmpl` extension, customize the hook's contents, and ensure that the script is executable. Windows, however, uses file extensions to determine whether or not a program is executable, so you would need to supply a program whose basename is the name of the hook, and whose extension is one of the special extensions recognized by Windows for executable programs, such as `.exe` for programs, and `.bat` for batch files.

For security reasons, the Subversion repository executes hook programs with an empty environment—that is, no environment variables are set at all, not even `$PATH` (or `%PATH%`, under Windows). Because of this, many administrators are baffled when their hook program runs fine by hand, but doesn't work when run by Subversion. Be sure to explicitly set any necessary environment variables in your hook program and/or use absolute paths to programs.

Subversion executes hooks as the same user who owns the process which is accessing the Subversion repository. In most cases, the repository is being accessed via a Subversion server, so this user is the same user as which that server runs on the system. The hooks themselves will need to be configured with OS-level permissions that allow that user to execute them. Also, this means that any file or programs (including the Subversion repository itself) accessed directly or indirectly by the hook will be accessed as the same user. In other words, be alert to potential permission-related problems that could prevent the hook from performing the tasks it is designed to perform.

There are nine hooks implemented by the Subversion repository, and you can get details about each of them in “Repository Hooks”. As a repository administrator, you'll need to decide which of hooks you wish to implement (by way of providing an appropriately named and permissioned hook program), and how. When you make this decision, keep in mind the big picture of how your repository is deployed.

For example, if you are using server configuration to determine which users are permitted to commit changes to your repository, then you don't need to do this sort of access control via the hook system.

There is no shortage of Subversion hook programs and scripts freely available either from the Subversion community itself or elsewhere. These scripts cover a wide range of utility—basic access control, policy adherence checking, issue tracker integration, email- or syndication-based commit notification, and beyond. See Apêndice D, *Ferramentas de Terceiros* for discussion of some of the most commonly used hook programs. Or, if you wish to write your own, see Capítulo 8, *Incorporando o Subversion*.

While hook scripts can do almost anything, there is one dimension in which hook script authors should show restraint: do *not* modify a commit transaction using hook scripts. While it might be tempting to use hook scripts to automatically correct errors or shortcomings or policy violations present in the files being committed, doing so can cause problems. Subversion keeps client-side caches of certain bits of repository data, and if you change a commit transaction in this way, those caches become indetectably stale. This inconsistency can lead to surprising and unexpected behavior. Instead of modifying the transaction, you should simply *validate* the transaction in the `pre-commit` hook and reject the commit if it does not meet the desired requirements. As a bonus, your users will learn the value of careful, compliance-minded work habits.

Berkeley DB Configuration

A Berkeley DB environment is an encapsulation of one or more databases, log files, region files and configuration files. The Berkeley DB environment has its own set of default configuration values for things like the number of database locks allowed to be taken out at any given time, or the maximum size of the journaling log files, etc. Subversion's filesystem logic additionally chooses default values for some of the Berkeley DB configuration options. However, sometimes your particular repository, with its unique collection of data and access patterns, might require a different set of configuration option values.

The producers of Berkeley DB understand that different applications and database environments have different requirements, and so they have provided a mechanism for overriding at runtime many of the configuration values for the Berkeley DB environment: BDB checks for the presence of a file named `DB_CONFIG` in the environment directory (namely, the repository's `db` subdirectory), and parses the options found in that file . Subversion itself creates this file when it creates the rest of the repository. The file initially contains some default options, as well as pointers to the Berkeley DB online documentation so you can read about what those options do. Of course, you are free to add any of the supported Berkeley DB options to your `DB_CONFIG` file. Just be aware that while Subversion never attempts to read or interpret the contents of the file, and makes no direct use of the option settings in it, you'll want to avoid any configuration changes that may cause Berkeley DB to behave in a fashion that is at odds with what Subversion might expect. Also, changes made to `DB_CONFIG` won't take effect until you recover the database environment (using **svnadmin recover**).

Repository Maintenance

Maintaining a Subversion repository can be daunting, mostly due to the complexities inherent in systems which have a database backend. Doing the task well is all about knowing the tools—what they are, when to use them, and how to use them. This section will introduce you to the repository administration tools provided by Subversion, and how to wield them to accomplish tasks such as repository data migration, upgrades, backups and cleanups.

An Administrator's Toolkit

Subversion provides a handful of utilities useful for creating, inspecting, modifying and repairing your repository. Let's look more closely at each of those tools. Afterward, we'll briefly examine some of the utilities included in the Berkeley DB distribution that provide functionality specific to your repository's database backend not otherwise provided by Subversion's own tools.

svnadmin

The **svnadmin** program is the repository administrator's best friend. Besides providing the ability to create Subversion repositories, this program allows you to perform several maintenance operations on those repositories. The syntax of **svnadmin** is similar to that of other Subversion command-line programs:

```
$ svnadmin help
general usage: svnadmin SUBCOMMAND REPOS_PATH  [ARGS & OPTIONS ...]
Type 'svnadmin help <subcommand>' for help on a specific subcommand.
Type 'svnadmin --version' to see the program version and FS modules.

Available subcommands:
   crashtest
   create
   deltify
…
```

We've already mentioned **svnadmin**'s `create` subcommand (see “Creating the Repository”). Most of the others we will cover later in this chapter. And you can consult “svnadmin” for a full rundown of subcommands and what each of them offers.

svnlook

svnlook is a tool provided by Subversion for examining the various revisions and *transactions* (which are revisions in-the-making) in a repository. No part of this program attempts to change the repository. **svnlook** is typically used by the repository hooks for reporting the changes that are about to be committed (in the case of the **pre-commit** hook) or that were just committed (in the case of the **post-commit** hook) to the repository. A repository administrator may use this tool for diagnostic purposes.

svnlook has a straightforward syntax:

```
$ svnlook help
general usage: svnlook SUBCOMMAND REPOS_PATH [ARGS & OPTIONS ...]
Note: any subcommand which takes the '--revision' and '--transaction'
      options will, if invoked without one of those options, act on
      the repository's youngest revision.
Type 'svnlook help <subcommand>' for help on a specific subcommand.
Type 'svnlook --version' to see the program version and FS modules.
…
```

Nearly every one of **svnlook**'s subcommands can operate on either a revision or a transaction tree, printing information about the tree itself, or how it differs from the previous revision of the repository. You use the `--revision (-r)` and `--transaction (-t)` options to specify which revision or transaction, respectively, to examine. In the absence of both the `--revision (-r)` and `--transaction (-t)` options, **svnlook** will examine the youngest (or “HEAD”) revision in the repository. So the following two commands do exactly the same thing when 19 is the youngest revision in the repository located at `/path/to/repos`:

```
$ svnlook info /path/to/repos
$ svnlook info /path/to/repos -r 19
```

The only exception to these rules about subcommands is the **svnlook youngest** subcommand, which takes no options, and simply prints out the repository's youngest revision number.

```
$ svnlook youngest /path/to/repos
19
```

Keep in mind that the only transactions you can browse are uncommitted ones. Most repositories will have no such transactions, because transactions are usually either committed (in which case, you should access them as revision with the `--revision (-r)` option) or aborted and removed.

Output from **svnlook** is designed to be both human- and machine-parsable. Take as an example the output of the `info` subcommand:

```
$ svnlook info /path/to/repos
sally
2002-11-04 09:29:13 -0600 (Mon, 04 Nov 2002)
27
Added the usual
Greek tree.
```

The output of the `info` subcommand is defined as:

1. The author, followed by a newline.
2. The date, followed by a newline.
3. The number of characters in the log message, followed by a newline.
4. The log message itself, followed by a newline.

This output is human-readable, meaning items like the datestamp are displayed using a textual representation instead of something more obscure (such as the number of nanoseconds since the Tasty Freeze guy drove by). But the output is also machine-parsable—because the log message can contain multiple lines and be unbounded in length, **svnlook** provides the length of that message before the message itself. This allows scripts and other wrappers around this command to make intelligent decisions about the log message, such as how much memory to allocate for the message, or at least how many bytes to skip in the event that this output is not the last bit of data in the stream.

svnlook can perform a variety of other queries: displaying subsets of bits of information we've mentioned previously, recursively listing versioned directory trees, reporting which paths were modified in a given revision or transaction, showing textual and property differences made to files and directories, and so on. See “svnlook” for a full reference of **svnlook**'s features.

svndumpfilter

While it won't be the most commonly used tool at the administrator's disposal, **svndumpfilter** provides a very particular brand of useful functionality—the ability to quickly and easily modify streams of Subversion repository history data by acting as a path-based filter.

The syntax of **svndumpfilter** is as follows:

```
$ svndumpfilter help
general usage: svndumpfilter SUBCOMMAND [ARGS & OPTIONS ...]
Type "svndumpfilter help <subcommand>" for help on a specific subcommand.
Type 'svndumpfilter --version' to see the program version.

Available subcommands:
   exclude
   include
   help (?, h)
```

There are only two interesting subcommands. They allow you to make the choice between explicit or implicit inclusion of paths in the stream:

`exclude`
: Filter out a set of paths from the dump data stream.

`include`
: Allow only the requested set of paths to pass through the dump data stream.

You can learn more about these subcommands and **svndumpfilter**'s unique purpose in “Filtering Repository History”.

svnsync

The **svnsync** program, which is new to the 1.4 release of Subversion, provides all the functionality required for maintaining a read-only mirror of a Subversion repository. The program really has one job —to transfer one repository's versioned history into another repository. And while there are few ways to do that, its primary strength is that it can operate remotely—the “source” and “sink” [7] repositories may be on different computers from each other and from **svnsync** itself.

As you might expect, **svnsync** has a syntax that looks very much like every other program we've mentioned in this chapter:

```
$ svnsync help
general usage: svnsync SUBCOMMAND DEST_URL  [ARGS & OPTIONS ...]
Type 'svnsync help <subcommand>' for help on a specific subcommand.
Type 'svnsync --version' to see the program version and RA modules.

Available subcommands:
   initialize (init)
   synchronize (sync)
   copy-revprops
   help (?, h)
$
```

We talk more about replication repositories with **svnsync** in “Repository Replication”.

Berkeley DB Utilities

If you're using a Berkeley DB repository, then all of your versioned filesystem's structure and data live in a set of database tables within the `db/` subdirectory of your repository. This subdirectory is a regular Berkeley DB environment directory, and can therefore be used in conjunction with any of the Berkeley database tools, typically provided as part of the Berkeley DB distribution.

For day-to-day Subversion use, these tools are unnecessary. Most of the functionality typically needed for Subversion repositories has been duplicated in the **svnadmin** tool. For example, **svnadmin list-unused-dblogs** and **svnadmin list-dblogs** perform a subset of what is provided by the Berkeley **db_archive** command, and **svnadmin recover** reflects the common use cases of the **db_recover** utility.

However, there are still a few Berkeley DB utilities that you might find useful. The **db_dump** and **db_load** programs write and read, respectively, a custom file format which describes the keys and values in a Berkeley DB database. Since Berkeley databases are not portable across machine architectures, this format is a useful way to transfer those databases from machine to machine, irrespective of architecture or operating system. As we describe later in this chapter, you can also use **svnadmin dump** and **svnadmin load** for similar purposes, but **db_dump** and **db_load** can do certain jobs just as well and much faster. They can also be useful if the experienced Berkeley DB hacker needs to do in-place

[7] Or is that, the “sync”?

tweaking of the data in a BDB-backed repository for some reason, which is something Subversion's utilities won't allow. Also, the **db_stat** utility can provide useful information about the status of your Berkeley DB environment, including detailed statistics about the locking and storage subsystems.

For more information on the Berkeley DB tool chain, visit the documentation section of the Berkeley DB section of Oracle's website, located at http://www.oracle.com/technology/documentation/berkeley-db/db/.

Commit Log Message Correction

Sometimes a user will have an error in her log message (a misspelling or some misinformation, perhaps). If the repository is configured (using the `pre-revprop-change` hook; see "Implementing Repository Hooks") to accept changes to this log message after the commit is finished, then the user can "fix" her log message remotely using the **svn** program's `propset` command (see svn propset). However, because of the potential to lose information forever, Subversion repositories are not, by default, configured to allow changes to unversioned properties—except by an administrator.

If a log message needs to be changed by an administrator, this can be done using **svnadmin setlog**. This command changes the log message (the `svn:log` property) on a given revision of a repository, reading the new value from a provided file.

```
$ echo "Here is the new, correct log message" > newlog.txt
$ svnadmin setlog myrepos newlog.txt -r 388
```

The **svnadmin setlog** command, by default, is still bound by the same protections against modifying unversioned properties as a remote client is—the `pre-` and `post-revprop-change` hooks are still triggered, and therefore must be set up to accept changes of this nature. But an administrator can get around these protections by passing the `--bypass-hooks` option to the **svnadmin setlog** command.

Remember, though, that by bypassing the hooks, you are likely avoiding such things as email notifications of property changes, backup systems which track unversioned property changes, and so on. In other words, be very careful about what you are changing, and how you change it.

Managing Disk Space

While the cost of storage has dropped incredibly in the past few years, disk usage is still a valid concern for administrators seeking to version large amounts of data. Every bit of version history information stored in the live repository needs to be backed up elsewhere, perhaps multiple times as part of rotating backup schedules. It is useful to know what pieces of Subversion's repository data need to remain on the live site, which need to be backed up, and which can be safely removed.

How Subversion saves disk space

To keep the repository small, Subversion uses *deltification* (or, "deltified storage") within the repository itself. Deltification involves encoding the representation of a chunk of data as a collection of differences against some other chunk of data. If the two pieces of data are very similar, this deltification results in storage savings for the deltified chunk—rather than taking up space equal to the size of the original data, it takes up only enough space to say, "I look just like this other piece of data over here, except for the following couple of changes". The result is that most of the repository data that tends to be bulky—namely, the contents of versioned files—is stored at a much smaller size than the original "fulltext" representation of that data. And for repositories created with Subversion 1.4 or later, the space savings are even better—now those fulltext representations of file contents are themselves compressed.

Because all of the data that is subject to deltification in a BDB-backed repository is stored in a single Berkeley DB database file, reducing the size of the stored values will not immediately reduce the size of the database file itself. Berkeley DB will, however, keep internal records of unused areas of the database file, and consume those areas first before

growing the size of the database file. So while deltification doesn't produce immediate space savings, it can drastically slow future growth of the database.

Removing dead transactions

Though they are uncommon, there are circumstances in which a Subversion commit process might fail, leaving behind in the repository the remnants of the revision-to-be that wasn't—an uncommitted transaction and all the file and directory changes associated with it. This could happen for several reasons: perhaps the client operation was inelegantly terminated by the user, or a network failure occurred in the middle of an operation. Regardless of the reason, dead transactions can happen. They don't do any real harm, other than consuming disk space. A fastidious administrator may nonetheless wish to remove them.

You can use **svnadmin**'s `lstxns` command to list the names of the currently outstanding transactions.

```
$ svnadmin lstxns myrepos
19
3a1
a45
$
```

Each item in the resultant output can then be used with **svnlook** (and its `--transaction (-t)` option) to determine who created the transaction, when it was created, what types of changes were made in the transaction—information that is helpful in determining whether or not the transaction is a safe candidate for removal! If you do indeed want to remove a transaction, its name can be passed to **svnadmin rmtxns**, which will perform the cleanup of the transaction. In fact, the `rmtxns` subcommand can take its input directly from the output of `lstxns`!

```
$ svnadmin rmtxns myrepos `svnadmin lstxns myrepos`
$
```

If you use these two subcommands like this, you should consider making your repository temporarily inaccessible to clients. That way, no one can begin a legitimate transaction before you start your cleanup. Exemplo 5.1, "txn-info.sh (Reporting Outstanding Transactions)" contains a bit of shell-scripting that can quickly generate information about each outstanding transaction in your repository.

Exemplo 5.1. txn-info.sh (Reporting Outstanding Transactions)

```
#!/bin/sh

### Generate informational output for all outstanding transactions in
### a Subversion repository.

REPOS="${1}"
if [ "x$REPOS" = x ] ; then
  echo "usage: $0 REPOS_PATH"
  exit
fi

for TXN in `svnadmin lstxns ${REPOS}`; do
  echo "---[ Transaction ${TXN} ]-------------------------------------------"
  svnlook info "${REPOS}" -t "${TXN}"
done
```

The output of the script is basically a concatenation of several chunks of **svnlook info** output (see "svnlook"), and will look something like:

```
$ txn-info.sh myrepos
---[ Transaction 19 ]-------------------------------------------
sally
2001-09-04 11:57:19 -0500 (Tue, 04 Sep 2001)
0
---[ Transaction 3a1 ]-------------------------------------------
harry
2001-09-10 16:50:30 -0500 (Mon, 10 Sep 2001)
39
Trying to commit over a faulty network.
---[ Transaction a45 ]-------------------------------------------
sally
2001-09-12 11:09:28 -0500 (Wed, 12 Sep 2001)
0
$
```

A long-abandoned transaction usually represents some sort of failed or interrupted commit. A transaction's datestamp can provide interesting information—for example, how likely is it that an operation begun nine months ago is still active?

In short, transaction cleanup decisions need not be made unwisely. Various sources of information—including Apache's error and access logs, Subversion's operational logs, Subversion revision history, and so on—can be employed in the decision-making process. And of course, an administrator can often simply communicate with a seemingly dead transaction's owner (via email, for example) to verify that the transaction is, in fact, in a zombie state.

Purging unused Berkeley DB logfiles

Until recently, the largest offender of disk space usage with respect to BDB-backed Subversion repositories was the log files in which Berkeley DB performs its pre-writes before modifying the actual database files. These files capture all the actions taken along the route of changing the database from one state to another—while the database files, at any given time, reflect a particular state, the log files contain all the many changes along the way *between* states. Thus, they can grow and accumulate quite rapidly.

Fortunately, beginning with the 4.2 release of Berkeley DB, the database environment has the ability to remove its own unused log files automatically. Any repositories created using an **svnadmin** which is compiled against Berkeley DB version 4.2 or greater will be configured for this automatic log file removal. If you don't want this feature enabled, simply pass the `--bdb-log-keep` option to the **svnadmin create** command. If you forget to do this, or change your mind at a later time, simply edit the `DB_CONFIG` file found in your repository's `db` directory, comment out the line which contains the `set_flags DB_LOG_AUTOREMOVE` directive, and then run **svnadmin recover** on your repository to force the configuration changes to take effect. See “Berkeley DB Configuration” for more information about database configuration.

Without some sort of automatic log file removal in place, log files will accumulate as you use your repository. This is actually somewhat of a feature of the database system—you should be able to recreate your entire database using nothing but the log files, so these files can be useful for catastrophic database recovery. But typically, you'll want to archive the log files that are no longer in use by Berkeley DB, and then remove them from disk to conserve space. Use the **svnadmin list-unused-dblogs** command to list the unused log files:

```
$ svnadmin list-unused-dblogs /path/to/repos
/path/to/repos/log.0000000031
/path/to/repos/log.0000000032
/path/to/repos/log.0000000033
…
```

```
$ rm `svnadmin list-unused-dblogs /path/to/repos`
## disk space reclaimed!
```

BDB-backed repositories whose log files are used as part of a backup or disaster recovery plan should *not* make use of the log file autoremoval feature. Reconstruction of a repository's data from log files can only be accomplished when *all* the log files are available. If some of the log files are removed from disk before the backup system has a chance to copy them elsewhere, the incomplete set of backed-up log files is essentially useless.

Berkeley DB Recovery

As mentioned in “Berkeley DB”, a Berkeley DB repository can sometimes be left in frozen state if not closed properly. When this happens, an administrator needs to rewind the database back into a consistent state. This is unique to BDB-backed repositories, though—if you are using FSFS-backed ones instead, this won't apply to you. And for those of you using Subversion 1.4 with Berkeley DB 4.4 or better, you should find that Subversion has become much more resilient in these types of situations. Still, wedged Berkeley DB repositories do occur, and an administrator needs to know how to safely deal with this circumstance.

In order to protect the data in your repository, Berkeley DB uses a locking mechanism. This mechanism ensures that portions of the database are not simultaneously modified by multiple database accessors, and that each process sees the data in the correct state when that data is being read from the database. When a process needs to change something in the database, it first checks for the existence of a lock on the target data. If the data is not locked, the process locks the data, makes the change it wants to make, and then unlocks the data. Other processes are forced to wait until that lock is removed before they are permitted to continue accessing that section of the database. (This has nothing to do with the locks that you, as a user, can apply to versioned files within the repository; we try to clear up the confusion caused by this terminology collision in Os três significados de trava.)

In the course of using your Subversion repository, fatal errors or interruptions can prevent a process from having the chance to remove the locks it has placed in the database. The result is that the back-end database system gets “wedged”. When this happens, any attempts to access the repository hang indefinitely (since each new accessor is waiting for a lock to go away—which isn't going to happen).

If this happens to your repository, don't panic. The Berkeley DB filesystem takes advantage of database transactions and checkpoints and pre-write journaling to ensure that only the most catastrophic of events [8] can permanently destroy a database environment. A sufficiently paranoid repository administrator will have made off-site backups of the repository data in some fashion, but don't head off to the tape backup storage closet just yet.

Instead, use the following recipe to attempt to “unwedge” your repository:

1. Make sure that there are no processes accessing (or attempting to access) the repository. For networked repositories, this means shutting down the Apache HTTP Server or svnserve daemon, too.

2. Become the user who owns and manages the repository. This is important, as recovering a repository while running as the wrong user can tweak the permissions of the repository's files in such a way that your repository will still be inaccessible even after it is “unwedged”.

3. Run the command **svnadmin recover /path/to/repos**. You should see output like this:

```
Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 19.
```

[8] E.g.: hard drive + huge electromagnet = disaster.

This command may take many minutes to complete.

4. Restart the server process.

This procedure fixes almost every case of repository lock-up. Make sure that you run this command as the user that owns and manages the database, not just as `root`. Part of the recovery process might involve recreating from scratch various database files (shared memory regions, for example). Recovering as `root` will create those files such that they are owned by `root`, which means that even after you restore connectivity to your repository, regular users will be unable to access it.

If the previous procedure, for some reason, does not successfully unwedge your repository, you should do two things. First, move your broken repository directory aside (perhaps by renaming it to something like `repos.BROKEN`) and then restore your latest backup of it. Then, send an email to the Subversion user list (at <`users@subversion.tigris.org`>) describing your problem in detail. Data integrity is an extremely high priority to the Subversion developers.

Migrating Repository Data Elsewhere

A Subversion filesystem has its data spread throughout files in the repository, in a fashion generally understood by (and of interest to) only the Subversion developers themselves. However, circumstances may arise that call for all, or some subset, of that data to be copied or moved into another repository.

Subversion provides such functionality by way of repository dump streams. A repository dump stream (often referred to as a “dumpfile” when stored as a file on disk) is a portable, flat file format that describes the various revisions in your repository—what was changed, by whom, when, and so on. This dump stream is the primary mechanism used to marshal versioned history—in whole or in part, with or without modification—between repositories. And Subversion provides the tools necessary for creating and loading these dump streams—the **svnadmin dump** and **svnadmin load** subcommands, respectively.

While the Subversion repository dump format contains human-readable portions and a familiar structure (it resembles an RFC-822 format, the same type of format used for most email), it is *not* a plaintext file format. It is a binary file format, highly sensitive to meddling. For example, many text editors will corrupt the file by automatically converting line endings.

There are many reasons for dumping and loading Subversion repository data. Early in Subversion's life, the most common reason was due to the evolution of Subversion itself. As Subversion matured, there were times when changes made to the back-end database schema caused compatibility issues with previous versions of the repository, so users had to dump their repository data using the previous version of Subversion, and load it into a freshly created repository with the new version of Subversion. Now, these types of schema changes haven't occurred since Subversion's 1.0 release, and the Subversion developers promise not to force users to dump and load their repositories when upgrading between minor versions (such as from 1.3 to 1.4) of Subversion. But there are still other reasons for dumping and loading, including re-deploying a Berkeley DB repository on a new OS or CPU architecture, switching between the Berkeley DB and FSFS back-ends, or (as we'll cover in “Filtering Repository History”) purging versioned data from repository history.

Whatever your reason for migrating repository history, using the **svnadmin dump** and **svnadmin load** subcommands is straightforward. **svnadmin dump** will output a range of repository revisions that are formatted using Subversion's custom filesystem dump format. The dump format is printed to the standard output stream, while informative messages are printed to the standard error stream. This allows you to redirect the output stream to a file while watching the status output in your terminal window. For example:

```
$ svnlook youngest myrepos
26
$ svnadmin dump myrepos > dumpfile
* Dumped revision 0.
```

```
* Dumped revision 1.
* Dumped revision 2.
…
* Dumped revision 25.
* Dumped revision 26.
```

At the end of the process, you will have a single file (`dumpfile` in the previous example) that contains all the data stored in your repository in the requested range of revisions. Note that **svnadmin dump** is reading revision trees from the repository just like any other “reader” process would (**svn checkout**, for example), so it's safe to run this command at any time.

The other subcommand in the pair, **svnadmin load**, parses the standard input stream as a Subversion repository dump file, and effectively replays those dumped revisions into the target repository for that operation. It also gives informative feedback, this time using the standard output stream:

```
$ svnadmin load newrepos < dumpfile
<<< Started new txn, based on original revision 1
     * adding path : A ... done.
     * adding path : A/B ... done.
     …
------- Committed new rev 1 (loaded from original rev 1) >>>

<<< Started new txn, based on original revision 2
     * editing path : A/mu ... done.
     * editing path : A/D/G/rho ... done.

------- Committed new rev 2 (loaded from original rev 2) >>>

…

<<< Started new txn, based on original revision 25
     * editing path : A/D/gamma ... done.

------- Committed new rev 25 (loaded from original rev 25) >>>

<<< Started new txn, based on original revision 26
     * adding path : A/Z/zeta ... done.
     * editing path : A/mu ... done.

------- Committed new rev 26 (loaded from original rev 26) >>>
```

The result of a load is new revisions added to a repository—the same thing you get by making commits against that repository from a regular Subversion client. And just as in a commit, you can use hook programs to perform actions before and after each of the commits made during a load process. By passing the `--use-pre-commit-hook` and `--use-post-commit-hook` options to **svnadmin load**, you can instruct Subversion to execute the pre-commit and post-commit hook programs, respectively, for each loaded revision. You might use these, for example, to ensure that loaded revisions pass through the same validation steps that regular commits pass through. Of course, you should use these options with care—if your post-commit hook sends emails to a mailing list for each new commit, you might not want to spew hundreds or thousands of commit emails in rapid succession at that list! You can read more about the use of hook scripts in “Implementing Repository Hooks”.

Note that because **svnadmin** uses standard input and output streams for the repository dump and load process, people who are feeling especially saucy can try things like this (perhaps even using different versions of **svnadmin** on each side of the pipe):

```
$ svnadmin create newrepos
$ svnadmin dump oldrepos | svnadmin load newrepos
```

By default, the dump file will be quite large—much larger than the repository itself. That's because by default every version of every file is expressed as a full text in the dump file. This is the fastest and simplest behavior, and nice if you're piping the dump data directly into some other process (such as a compression program, filtering program, or into a loading process). But if you're creating a dump file for longer-term storage, you'll likely want to save disk space by using the `--deltas` option. With this option, successive revisions of files will be output as compressed, binary differences—just as file revisions are stored in a repository. This option is slower, but results in a dump file much closer in size to the original repository.

We mentioned previously that **svnadmin dump** outputs a range of revisions. Use the `--revision (-r)` option to specify a single revision to dump, or a range of revisions. If you omit this option, all the existing repository revisions will be dumped.

```
$ svnadmin dump myrepos -r 23 > rev-23.dumpfile
$ svnadmin dump myrepos -r 100:200 > revs-100-200.dumpfile
```

As Subversion dumps each new revision, it outputs only enough information to allow a future loader to re-create that revision based on the previous one. In other words, for any given revision in the dump file, only the items that were changed in that revision will appear in the dump. The only exception to this rule is the first revision that is dumped with the current **svnadmin dump** command.

By default, Subversion will not express the first dumped revision as merely differences to be applied to the previous revision. For one thing, there is no previous revision in the dump file! And secondly, Subversion cannot know the state of the repository into which the dump data will be loaded (if it ever is). To ensure that the output of each execution of **svnadmin dump** is self-sufficient, the first dumped revision is by default a full representation of every directory, file, and property in that revision of the repository.

However, you can change this default behavior. If you add the `--incremental` option when you dump your repository, **svnadmin** will compare the first dumped revision against the previous revision in the repository, the same way it treats every other revision that gets dumped. It will then output the first revision exactly as it does the rest of the revisions in the dump range—mentioning only the changes that occurred in that revision. The benefit of this is that you can create several small dump files that can be loaded in succession, instead of one large one, like so:

```
$ svnadmin dump myrepos -r 0:1000 > dumpfile1
$ svnadmin dump myrepos -r 1001:2000 --incremental > dumpfile2
$ svnadmin dump myrepos -r 2001:3000 --incremental > dumpfile3
```

These dump files could be loaded into a new repository with the following command sequence:

```
$ svnadmin load newrepos < dumpfile1
$ svnadmin load newrepos < dumpfile2
$ svnadmin load newrepos < dumpfile3
```

Another neat trick you can perform with this `--incremental` option involves appending to an existing dump file a new range of dumped revisions. For example, you might have a `post-commit` hook that simply appends the repository dump of the single revision that triggered the hook. Or you might have a script that runs nightly to append dump file data for all the revisions that were added to the repository since the last time the script ran. Used like this, **svnadmin dump** can be one way to back up changes to your repository over time in case of a system crash or some other catastrophic event.

The dump format can also be used to merge the contents of several different repositories into a single repository. By using the `--parent-dir` option of **svnadmin load**, you can specify a new virtual root

directory for the load process. That means if you have dump files for three repositories, say `calc-dumpfile`, `cal-dumpfile`, and `ss-dumpfile`, you can first create a new repository to hold them all:

```
$ svnadmin create /path/to/projects
$
```

Then, make new directories in the repository which will encapsulate the contents of each of the three previous repositories:

```
$ svn mkdir -m "Initial project roots" \
      file:///path/to/projects/calc \
      file:///path/to/projects/calendar \
      file:///path/to/projects/spreadsheet
Committed revision 1.
$
```

Lastly, load the individual dump files into their respective locations in the new repository:

```
$ svnadmin load /path/to/projects --parent-dir calc < calc-dumpfile
…
$ svnadmin load /path/to/projects --parent-dir calendar < cal-dumpfile
…
$ svnadmin load /path/to/projects --parent-dir spreadsheet < ss-dumpfile
…
$
```

We'll mention one final way to use the Subversion repository dump format—conversion from a different storage mechanism or version control system altogether. Because the dump file format is, for the most part, human-readable, it should be relatively easy to describe generic sets of changes—each of which should be treated as a new revision—using this file format. In fact, the **cvs2svn** utility (see “Convertendo um Repositório de CVS para Subversion”) uses the dump format to represent the contents of a CVS repository so that those contents can be copied into a Subversion repository.

Filtering Repository History

Since Subversion stores your versioned history using, at the very least, binary differencing algorithms and data compression (optionally in a completely opaque database system), attempting manual tweaks is unwise, if not quite difficult, and at any rate strongly discouraged. And once data has been stored in your repository, Subversion generally doesn't provide an easy way to remove that data. [9] But inevitably, there will be times when you would like to manipulate the history of your repository. You might need to strip out all instances of a file that was accidentally added to the repository (and shouldn't be there for whatever reason). [10] Or, perhaps you have multiple projects sharing a single repository, and you decide to split them up into their own repositories. To accomplish tasks like this, administrators need a more manageable and malleable representation of the data in their repositories—the Subversion repository dump format.

As we described in “Migrating Repository Data Elsewhere”, the Subversion repository dump format is a human-readable representation of the changes that you've made to your versioned data over time. You use the **svnadmin dump** command to generate the dump data, and **svnadmin load** to populate a new repository with it (see “Migrating Repository Data Elsewhere”). The great thing about the human-readability aspect of the dump format is that, if you aren't careless about it, you can manually inspect and modify it. Of course, the downside is that if you have three years' worth of repository activity encapsulated

[9] That's rather the reason you use version control at all, right?

[10] Conscious, cautious removal of certain bits of versioned data is actually supported by real use-cases. That's why an “obliterate” feature has been one of the most highly requested Subversion features, and one which the Subversion developers hope to soon provide.

in what is likely to be a very large dump file, it could take you a long, long time to manually inspect and modify it.

That's where **svndumpfilter** becomes useful. This program acts as path-based filter for repository dump streams. Simply give it either a list of paths you wish to keep, or a list of paths you wish to not keep, then pipe your repository dump data through this filter. The result will be a modified stream of dump data that contains only the versioned paths you (explicitly or implicitly) requested.

Let's look a realistic example of how you might use this program. We discuss elsewhere (see “Planejando a Organização do Repositório”) the process of deciding how to choose a layout for the data in your repositories—using one repository per project or combining them, arranging stuff within your repository, and so on. But sometimes after new revisions start flying in, you rethink your layout and would like to make some changes. A common change is the decision to move multiple projects which are sharing a single repository into separate repositories for each project.

Our imaginary repository contains three projects: `calc`, `calendar`, and `spreadsheet`. They have been living side-by-side in a layout like this:

```
/
   calc/
      trunk/
      branches/
      tags/
   calendar/
      trunk/
      branches/
      tags/
   spreadsheet/
      trunk/
      branches/
      tags/
```

To get these three projects into their own repositories, we first dump the whole repository:

```
$ svnadmin dump /path/to/repos > repos-dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
* Dumped revision 3.
…
$
```

Next, run that dump file through the filter, each time including only one of our top-level directories, and resulting in three new dump files:

```
$ svndumpfilter include calc < repos-dumpfile > calc-dumpfile
…
$ svndumpfilter include calendar < repos-dumpfile > cal-dumpfile
…
$ svndumpfilter include spreadsheet < repos-dumpfile > ss-dumpfile
…
$
```

At this point, you have to make a decision. Each of your dump files will create a valid repository, but will preserve the paths exactly as they were in the original repository. This means that even though you would have a repository solely for your `calc` project, that repository would still have a top-level

directory named `calc`. If you want your `trunk`, `tags`, and `branches` directories to live in the root of your repository, you might wish to edit your dump files, tweaking the `Node-path` and `Node-copyfrom-path` headers to no longer have that first `calc/` path component. Also, you'll want to remove the section of dump data that creates the `calc` directory. It will look something like:

```
Node-path: calc
Node-action: add
Node-kind: dir
Content-length: 0
```

If you do plan on manually editing the dump file to remove a top-level directory, make sure that your editor is not set to automatically convert end-of-line characters to the native format (e.g. \r\n to \n), as the content will then not agree with the metadata. This will render the dump file useless.

All that remains now is to create your three new repositories, and load each dump file into the right repository:

```
$ svnadmin create calc; svnadmin load calc < calc-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : button.c ... done.
…
$ svnadmin create calendar; svnadmin load calendar < cal-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : cal.c ... done.
…
$ svnadmin create spreadsheet; svnadmin load spreadsheet < ss-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : ss.c ... done.
…
$
```

Both of **svndumpfilter**'s subcommands accept options for deciding how to deal with “empty” revisions. If a given revision contained only changes to paths that were filtered out, that now-empty revision could be considered uninteresting or even unwanted. So to give the user control over what to do with those revisions, **svndumpfilter** provides the following command-line options:

`--drop-empty-revs`
: Do not generate empty revisions at all—just omit them.

`--renumber-revs`
: If empty revisions are dropped (using the `--drop-empty-revs` option), change the revision numbers of the remaining revisions so that there are no gaps in the numeric sequence.

`--preserve-revprops`
: If empty revisions are not dropped, preserve the revision properties (log message, author, date, custom properties, etc.) for those empty revisions. Otherwise, empty revisions will only contain the original datestamp, and a generated log message that indicates that this revision was emptied by **svndumpfilter**.

While **svndumpfilter** can be very useful, and a huge timesaver, there are unfortunately a couple of gotchas. First, this utility is overly sensitive to path semantics. Pay attention to whether paths in your dump file are specified with or without leading slashes. You'll want to look at the `Node-path` and `Node-copyfrom-path` headers.

```
…
Node-path: spreadsheet/Makefile
…
```

If the paths have leading slashes, you should include leading slashes in the paths you pass to **svndumpfilter include** and **svndumpfilter exclude** (and if they don't, you shouldn't). Further, if your dump file has an inconsistent usage of leading slashes for some reason, [11] you should probably normalize those paths so they all have, or lack, leading slashes.

Also, copied paths can give you some trouble. Subversion supports copy operations in the repository, where a new path is created by copying some already existing path. It is possible that at some point in the lifetime of your repository, you might have copied a file or directory from some location that **svndumpfilter** is excluding, to a location that it is including. In order to make the dump data self-sufficient, **svndumpfilter** needs to still show the addition of the new path—including the contents of any files created by the copy—and not represent that addition as a copy from a source that won't exist in your filtered dump data stream. But because the Subversion repository dump format only shows what was changed in each revision, the contents of the copy source might not be readily available. If you suspect that you have any copies of this sort in your repository, you might want to rethink your set of included/excluded paths, perhaps including the paths that served as sources of your troublesome copy operations, too.

Finally, **svndumpfilter** takes path filtering quite literally. If you are trying to copy the history of a project rooted at `trunk/my-project` and move it into a repository of its own, you would, of course, use the **svndumpfilter include** command to keep all the changes in and under `trunk/my-project`. But the resulting dump file makes no assumptions about the repository into which you plan to load this data. Specifically, the dump data might begin with the revision which added the `trunk/my-project` directory, but it will *not* contain directives which would create the `trunk` directory itself (because `trunk` doesn't match the include filter). You'll need to make sure that any directories which the new dump stream expect to exist actually do exist in the target repository before trying to load the stream into that repository.

Repository Replication

There are several scenarios in which it is quite handy to have a Subversion repository whose version history is exactly the same as some other repository's. Perhaps the most obvious one is the maintenance of a simple backup repository, used when the primary repository has become inaccessible due to a hardware failure, network outage, or other such annoyance. Other scenarios include deploying mirror repositories to distribute heavy Subversion load across multiple servers, use as a soft-upgrade mechanism, and so on.

As of version 1.4, Subversion provides a program for managing scenarios like these—**svnsync**. **svnsync** works by essentially asking the Subversion server to “replay” revisions, one at a time. It then uses that revision information to mimic a commit of the same to another repository. Neither repository needs to be locally accessible to machine on which **svnsync** is running—its parameters are repository URLs, and it does all its work through Subversion's repository access (RA) interfaces. All it requires is read access to the source repository and read/write access to the destination repository.

When using **svnsync** against a remote source repository, the Subversion server for that repository must be running Subversion version 1.4 or better.

Assuming you already have a source repository that you'd like to mirror, the next thing you need is an empty target repository which will actually serve as that mirror. This target repository can use either of the available filesystem data-store back-ends (see “Escolhendo uma Base de Dados”), but it must not yet have any version history in it. The protocol via which **svnsync** communicates revision information

[11] While **svnadmin dump** has a consistent leading slash policy—to not include them—other programs which generate dump data might not be so consistent.

is highly sensitive to mismatches between the versioned histories contained in the source and target repositories. For this reason, while **svnsync** cannot *demand* that the target repository be read-only, [12] allowing the revision history in the target repository to change by any mechanism other than the mirroring process is a recipe for disaster.

Do *not* modify a mirror repository in such a way as to cause its version history to deviate from that of the repository it mirrors. The only commits and revision property modifications that ever occur on that mirror repository should be those performed by the **svnsync** tool.

Another requirement of the target repository is that the **svnsync** process be allowed to modify certain revision properties. **svnsync** stores its bookkeeping information in special revision properties on revision 0 of the destination repository. Because **svnsync** works within the framework of that repository's hook system, the default state of the repository (which is to disallow revision property changes; see pre-revprop-change) is insufficient. You'll need to explicitly implement the pre-revprop-change hook, and your script must allow **svnsync** to set and change its special properties. With those provisions in place, you are ready to start mirroring repository revisions.

It's a good idea to implement authorization measures which allow your repository replication process to perform its tasks while preventing other users from modifying the contents of your mirror repository at all.

Let's walk through the use of **svnsync** in a somewhat typical mirroring scenario. We'll pepper this discourse with practical recommendations which you are free to disregard if they aren't required by or suitable for your environment.

As a service to the fine developers of our favorite version control system, we will be mirroring the public Subversion source code repository and exposing that mirror publicly on the Internet, hosted on a different machine than the one on which the original Subversion source code repository lives. This remote host has a global configuration which permits anonymous users to read the contents of repositories on the host, but requires users to authenticate in order to modify those repositories. (Please forgive us for glossing over the details of Subversion server configuration for the moment—those are covered thoroughly in Capítulo 6, *Configuração do Servidor*.) And for no other reason than that it makes for a more interesting example, we'll be driving the replication process from a third machine, the one which we currently find ourselves using.

First, we'll create the repository which will be our mirror. This and the next couple of steps do require shell access to the machine on which the mirror repository will live. Once the repository is all configured, though, we shouldn't need to touch it directly again.

```
$ ssh admin@svn.example.com \
      "svnadmin create /path/to/repositories/svn-mirror"
admin@svn.example.com's password: ********
$
```

At this point, we have our repository, and due to our server's configuration, that repository is now “live” on the Internet. Now, because we don't want anything modifying the repository except our replication process, we need a way to distinguish that process from other would-be committers. To do so, we use a dedicated username for our process. Only commits and revision property modifications performed by the special username `syncuser` will be allowed.

We'll use the repository's hook system both to allow the replication process to do what it needs to do, and to enforce that only it is doing those things. We accomplish this by implementing two of the repository event hooks—pre-revprop-change and start-commit. Our `pre-revprop-change` hook script is found in Exemplo 5.2, “Mirror repository's pre-revprop-change hook script”, and basically verifies that the user attempting the property changes is our `syncuser` user. If so, the change is allowed; otherwise, it is denied.

[12] In fact, it can't truly be read-only, or **svnsync** itself would have a tough time copying revision history into it.

Exemplo 5.2. Mirror repository's pre-revprop-change hook script

```
#!/bin/sh

USER="$3"

if [ "$USER" = "syncuser" ]; then exit 0; fi

echo "Only the syncuser user may change revision properties" >&2
exit 1
```

That covers revision property changes. Now we need to ensure that only the `syncuser` user is permitted to commit new revisions to the repository. We do this using a `start-commit` hook scripts like the one in Exemplo 5.3, “Mirror repository's start-commit hook script”.

Exemplo 5.3. Mirror repository's start-commit hook script

```
#!/bin/sh

USER="$2"

if [ "$USER" = "syncuser" ]; then exit 0; fi

echo "Only the syncuser user may commit new revisions" >&2
exit 1
```

After installing our hook scripts and ensuring that they are executable by the Subversion server, we're finished with the setup of the mirror repository. Now, we get to actually do the mirroring.

The first thing we need to do with **svnsync** is to register in our target repository the fact that it will be a mirror of the source repository. We do this using the **svnsync initialize** subcommand. Note that the various **svnsync** subcommands provide several of the same authentication-related options that **svn** does: `--username`, `--password`, `--non-interactive`, `--config-dir`, and `--no-auth-cache`.

```
$ svnsync help init
initialize (init): usage: svnsync initialize DEST_URL SOURCE_URL

Initialize a destination repository for synchronization from
another repository.

The destination URL must point to the root of a repository with
no committed revisions.  The destination repository must allow
revision property changes.

You should not commit to, or make revision property changes in,
the destination repository by any method other than 'svnsync'.
In other words, the destination repository should be a read-only
mirror of the source repository.

Valid options:
  --non-interactive        : do no interactive prompting
  --no-auth-cache          : do not cache authentication tokens
  --username arg           : specify a username ARG
  --password arg           : specify a password ARG
  --config-dir arg         : read user configuration files from directory ARG
```

```
$ svnsync initialize http://svn.example.com/svn-mirror \
                     http://svn.collab.net/repos/svn \
                     --username syncuser --password syncpass
Copied properties for revision 0.
$
```

Our target repository will now remember that it is a mirror of the public Subversion source code repository. Notice that we provided a username and password as arguments to **svnsync**—that was required by the pre-revprop-change hook on our mirror repository.

The URLs provided to **svnsync** must point to the root directories of the target and source repositories, respectively. The tool does not handle mirroring of repository subtrees.

The initial release of **svnsync** (in Subversion 1.4) has a small shortcoming—the values given to the `--username` and `--password` command-line options get used for authentication against both the source and destination repositories. Obviously, there's no guarantee that the synchronizing user's credentials are the same in both places. In the event that they are not the same, users trying to run **svnsync** in non-interactive mode (with the `--non-interactive` option) might experience problems.

And now comes the fun part. With a single subcommand, we can tell **svnsync** to copy all the as-yet-unmirrored revisions from the source repository to the target. [13] The **svnsync synchronize** subcommand will peek into the special revision properties previously stored on the target repository, and determine what repository it is mirroring and that the most recently mirrored revision was revision 0. Then it will query the source repository and determine what the latest revision in that repository is. Finally, it asks the source repository's server to start replaying all the revisions between 0 and that latest revision. As **svnsync** get the resulting response from the source repository's server, it begins forwarding those revisions to the target repository's server as new commits.

```
$ svnsync help synchronize
synchronize (sync): usage: svnsync synchronize DEST_URL

Transfer all pending revisions from source to destination.
…
$ svnsync synchronize http://svn.example.com/svn-mirror \
                      --username syncuser --password syncpass
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
Copied properties for revision 3.
…
Committed revision 23406.
Copied properties for revision 23406.
Committed revision 23407.
Copied properties for revision 23407.
Committed revision 23408.
Copied properties for revision 23408.
```

Of particular interest here is that for each mirrored revision, there is first a commit of that revision to the target repository, and then property changes follow. This is because the initial commit is performed by (and attributed to) the user `syncuser`, and datestamped with the time as of that revision's creation. Also, Subversion's underlying repository access interfaces don't provide a mechanism for setting arbitrary

[13]Be forewarned that while it will take only a few seconds for the average reader to parse this paragraph and the sample output which follows it, the actual time required to complete such a mirroring operation is, shall we say, quite a bit longer.

revision properties as part of a commit. So **svnsync** follows up with an immediate series of property modifications which copy all the revision properties found for that revision in the source repository into the target repository. This also has the effect of fixing the author and datestamp of the revision to match that of the source repository.

Also noteworthy is that **svnsync** performs careful bookkeeping that allows it to be safely interrupted and restarted without ruining the integrity of the mirrored data. If a network glitch occurs while mirroring a repository, simply repeat the **svnsync synchronize** command and it will happily pick up right where it left off. In fact, as new revisions appear in the source repository, this is exactly what you to do in order to keep your mirror up-to-date.

There is, however, one bit of inelegance in the process. Because Subversion revision properties can be changed at any time throughout the lifetime of the repository, and don't leave an audit trail that indicates when they were changed, replication processes have to pay special attention to them. If you've already mirrored the first 15 revisions of a repository and someone then changes a revision property on revision 12, **svnsync** won't know to go back and patch up its copy of revision 12. You'll need to tell it to do so manually by using (or with some additionally tooling around) the **svnsync copy-revprops** subcommand, which simply re-replicates all the revision properties for a particular revision.

```
$ svnsync help copy-revprops
copy-revprops: usage: svnsync copy-revprops DEST_URL REV

Copy all revision properties for revision REV from source to
destination.
…
$ svnsync copy-revprops http://svn.example.com/svn-mirror 12 \
                        --username syncuser --password syncpass
Copied properties for revision 12.
$
```

That's repository replication in a nutshell. You'll likely want some automation around such a process. For example, while our example was a pull-and-push setup, you might wish to have your primary repository push changes to one or more blessed mirrors as part of its post-commit and post-revprop-change hook implementations. This would enable the mirror to be up-to-date in as near to realtime as is likely possible.

Also, while it isn't very commonplace to do so, **svnsync** does gracefully mirror repositories in which the user as whom it authenticates only has partial read access. It simply copies only the bits of the repository that it is permitted to see. Obviously such a mirror is not useful as a backup solution.

As far as user interaction with repositories and mirrors goes, it *is* possible to have a single working copy that interacts with both, but you'll have to jump through some hoops to make it happen. First, you need to ensure that both the primary and mirror repositories have the same repository UUID (which is not the case by default). You can set the mirror repository's UUID by loading a dump file stub into it which contains the UUID of the primary repository, like so:

```
$ cat - <<EOF | svnadmin load --force-uuid dest
SVN-fs-dump-format-version: 2

UUID: 65390229-12b7-0310-b90b-f21a5aa7ec8e
EOF
$
```

Now that the two repositories have the same UUID, you can use **svn switch --relocate** to point your working copy to whichever of the repositories you wish to operate against, a process which is described in svn switch. There is a possible danger here, though, in that if the primary and mirror repositories aren't in close synchronization, a working copy up-to-date with, and pointing to, the primary repository will, if relocated to point to an out-of-date mirror, become confused about the apparent sudden loss of

revisions it fully expects to be present, and throws errors to that effect. If this occurs, you can relocate your working copy back to the primary repository and then either wait until the mirror repository is up-to-date, or backdate your working copy to a revision you know is present in the sync repository and then retry the relocation.

Finally, be aware that the revision-based replication provided by **svnsync** is only that—replication of revisions. It does not include such things as the hook implementations, repository or server configuration data, uncommitted transactions, or information about user locks on repository paths. Only information carried by the Subversion repository dump file format is available for replication.

Repository Backup

Despite numerous advances in technology since the birth of the modern computer, one thing unfortunately rings true with crystalline clarity—sometimes, things go very, very awry. Power outages, network connectivity dropouts, corrupt RAM and crashed hard drives are but a taste of the evil that Fate is poised to unleash on even the most conscientious administrator. And so we arrive at a very important topic—how to make backup copies of your repository data.

There are two types of backup methods available for Subversion repository administrators—full and incremental. A full backup of the repository involves squirreling away in one sweeping action all the information required to fully reconstruct that repository in the event of a catastrophe. Usually, it means, quite literally, the duplication of the entire repository directory (which includes either a Berkeley DB or FSFS environment). Incremental backups are lesser things, backups of only the portion of the repository data that has changed since the previous backup.

As far as full backups go, the naive approach might seem like a sane one, but unless you temporarily disable all other access to your repository, simply doing a recursive directory copy runs the risk of generating a faulty backup. In the case of Berkeley DB, the documentation describes a certain order in which database files can be copied that will guarantee a valid backup copy. A similar ordering exists for FSFS data. But you don't have to implement these algorithms yourself, because the Subversion development team has already done so. The **svnadmin hotcopy** command takes care of the minutia involved in making a hot backup of your repository. And its invocation is as trivial as Unix's **cp** or Windows' **copy** operations:

```
$ svnadmin hotcopy /path/to/repos /path/to/repos-backup
```

The resulting backup is a fully functional Subversion repository, able to be dropped in as a replacement for your live repository should something go horribly wrong.

When making copies of a Berkeley DB repository, you can even instruct **svnadmin hotcopy** to purge any unused Berkeley DB logfiles (see “Purging unused Berkeley DB logfiles”) from the original repository upon completion of the copy. Simply provide the `--clean-logs` option on the command-line.

```
$ svnadmin hotcopy --clean-logs /path/to/bdb-repos /path/to/bdb-repos-backup
```

Additional tooling around this command is available, too. The `tools/backup/` directory of the Subversion source distribution holds the **hot-backup.py** script. This script adds a bit of backup management atop **svnadmin hotcopy**, allowing you to keep only the most recent configured number of backups of each repository. It will automatically manage the names of the backed-up repository directories to avoid collisions with previous backups, and will “rotate off” older backups, deleting them so only the most recent ones remain. Even if you also have an incremental backup, you might want to run this program on a regular basis. For example, you might consider using **hot-backup.py** from a program scheduler (such as **cron** on Unix systems) which will cause it to run nightly (or at whatever granularity of Time you deem safe).

Some administrators use a different backup mechanism built around generating and storing repository dump data. We described in “Migrating Repository Data Elsewhere” how to use **svnadmin dump --**

incremental to perform an incremental backup of a given revision or range of revisions. And of course, there is a full backup variation of this achieved by omitting the `--incremental` option to that command. There is some value in these methods, in that the format of your backed-up information is flexible—it's not tied to a particular platform, versioned filesystem type, or release of Subversion or Berkeley DB. But that flexibility comes at a cost, namely that restoring that data can take a long time—longer with each new revision committed to your repository. Also, as is the case with so many of the various backup methods, revision property changes made to already-backed-up revisions won't get picked up by a non-overlapping, incremental dump generation. For these reasons, we recommend against relying solely on dump-based backup approaches.

As you can see, each of the various backup types and methods has its advantages and disadvantages. The easiest is by far the full hot backup, which will always result in a perfect working replica of your repository. Should something bad happen to your live repository, you can restore from the backup with a simple recursive directory copy. Unfortunately, if you are maintaining multiple backups of your repository, these full copies will each eat up just as much disk space as your live repository. Incremental backups, by contrast, tend to be quicker to generate and smaller to store. But the restoration process can be a pain, often involving applying multiple incremental backups. And other methods have their own peculiarities. Administrators need to find the balance between the cost of making the backup and the cost of restoring it.

The **svnsync** program (see “Repository Replication”) actually provides a rather handy middle-ground approach. If you are regularly synchronizing a read-only mirror with your main repository, then in a pinch, your read-only mirror is probably a good candidate for replacing that main repository if it falls over. The primary disadvantage of this method is that only the versioned repository data gets synchronized—repository configuration files, user-specified repository path locks, and other items which might live in the physical repository directory but not *inside* the repository's virtual versioned filesystem are not handled by svnsync.

In any backup scenario, repository administrators need to be aware of how modifications to unversioned revision properties affect their backups. Since these changes do not themselves generate new revisions, they will not trigger post-commit hooks, and may not even trigger the pre-revprop-change and post-revprop-change hooks. [14] And since you can change revision properties without respect to chronological order—you can change any revision's properties at any time—an incremental backup of the latest few revisions might not catch a property modification to a revision that was included as part of a previous backup.

Generally speaking, only the truly paranoid would need to backup their entire repository, say, every time a commit occurred. However, assuming that a given repository has some other redundancy mechanism in place with relatively fine granularity (like per-commit emails or incremental dumps), a hot backup of the database might be something that a repository administrator would want to include as part of a system-wide nightly backup. It's your data—protect it as much as you'd like.

Often, the best approach to repository backups is a diversified one which leverages combinations of the methods described here. The Subversion developers, for example, back up the Subversion source code repository nightly using **hot-backup.py** and an offsite **rsync** of those full backups; keep multiple archives of all the commit and property change notification emails; and have repository mirrors maintained by various volunteers using **svnsync**. Your solution might be similar, but should be catered to your needs and that delicate balance of convenience with paranoia. And whatever you do, validate your backups from time to time—what good is a spare tire that has a hole in it? While all of this might not save your hardware from the iron fist of Fate, [15] it should certainly help you recover from those trying times.

Sumário

Até agora você deve ter tido um entendimento de como criar, configurar e manter repositórios Subversion. Nós o introduzimos a várias ferramentas que o ajudarão nessa tarefa. Ao longo deste capítulo, nós mostramos obstáculos comuns e sugestões para evitá-los.

[14] **svnadmin setlog** can be called in a way that bypasses the hook interface altogether.
[15] You know—the collective term for all of her “fickle fingers”.

Tudo que falta para você é decidir que tipo de informação armazenar no seu repositório, e finalmente, como deixá-lo disponível na rede. O próximo capítulo é todo sobre rede.

Capítulo 6. Configuração do Servidor

Um repositório Subversion pode ser acessado simultaneamente por clientes executando na mesma máquina na qual o repositório se encontra usando o método `file://`. Mas a configuração típica do Subversion envolve ter-se uma única máquina servidora sendo acessada por clientes em computadores por todo um escritório—ou, talvez, por todo o mundo.

Este capítulo descreve como ter seu repositório Subversion acessível a partir da máquina onde estiver instalado para uso por clientes remotos. Vamos cobrir os mecanismos de servidor do Subversion disponíveis atualmente, discutindo a configuração e o uso de cada um. Depois de ler esta seção, você deve ser capaz de decidir qual configuração é a adequada às suas necessidades, e entender como habilitar tal configuração em seu servidor.

Visão Geral

O Subversion foi desenvolvido com uma camada de rede abstrata. Isso significa que um repositório pode ser acessado via programação por qualquer tipo de processo servidor, e a API de “acesso ao repositório” do cliente permite aos programadores escrever plugins que falem com protocolos de rede relevantes. Em teoria, o Subversion pode usar um número infinito de implementações de rede. Na prática, há apenas dois servidores até o momento em que este livro estava sendo escrito.

O Apache é um servidor web extremamente popular; usando o módulo **mod_dav_svn**, o Apache pode acessar um repositório e torná-lo disponível para os clientes através do protocolo WebDAV/DeltaV, que é uma extensão do HTTP. Como o Apache é um servidor web extremamente extensível, ele provê um conjunto de recursos “de graça”, tais como comunicação SSL criptografada, sistema de log, integração com diversos sistemas de autenticação de terceiros, além de navegação simplificada nos repositórios.

Por outro lado está o **svnserve**: um programa servidor pequeno e leve que conversa com os clientes por meio de um protocolo específico. Pelo fato de ter sido explicitamente desenvolvido para o Subversion e de manter informações de estado (diferentemente do HTTP), este seu protocolo permite operações de rede significativamente mais rápidas—ainda que ao custo de alguns recursos. Este protocolo só entende autenticação do tipo CRAM-MD5, não possui recursos de log, nem de navegação web nos repositórios, e não tem opção de criptografar o tráfego de rede. Mas é, no entanto, extremamente fácil de configurar e é quase sempre a melhor opção para pequenas equipes que ainda estão iniciando com o Subversion.

Uma terceira opção é o uso do **svnserve** através de uma conexão SSH. Por mais que este cenário ainda use o **svnserve**, ele difere um pouco no que diz respeito aos recursos de uma implantação **svnserve** tradicional. SSH é usado para criptografar toda a comunicação. O SSH também é usado exclusivamente para autenticação, então são necessárias contas reais no sistema do host servidor (diferentemente do uso do **svnserve** tradicional, que possui suas próprias contas de usuário particulares.) Finalmente, pelo fato desta configuração precisar que cada usuário dispare um processo temporário **svnserve** particular, esta opção é equivalente (do ponto de vista de permissões) a permitir total acesso de um grupo local de usuários no repositório por meio de URLs `file://`. Assim, controle de acesso com base em caminhos não faz sentido, já que cada usuário está acessando os arquivos da base de dados diretamente.

Aqui está um breve sumário destas três configurações típicas de servidor.

Tabela 6.1. Comparação das Opções para o Servidor Subversion

Característica	Apache + mod_dav_svn	svnserve	svnserve sobre SSH
Opções de autenticação	Autenticação básica HTTP(S), certificados X.509, LDAP, NTLM, ou quaisquer outros	CRAM-MD5	SSH

Característica	Apache + mod_dav_svn	svnserve	svnserve sobre SSH
	mecanismos disponíveis ao Apache httpd		
Opções para contas de usuários	arquivo 'users' privativo	arquivo 'users' privativo	contas no sistema
Opções de autorização	acesso leitura/escrita pode ser dado para o repositório como um todo, ou especificado por caminho	acesso leitura/escrita pode ser dado para o repositório como um todo, ou especificado por caminho	acesso leitura/escrita passível de ser dado apenas ao repositório como um todo
Criptografia	através de SSL opcional	nenhuma	túnel SSH
Registro de log	logs completos do Apache para cada requisição HTTP, com opcional log "alto nível" para operações do cliente em geral	sem log	sem log
Interoperabilidade	parcialmente usável por outros clientes WebDAV	se comunica apenas com clientes svn	se comunica apenas com clientes svn
Visualização pela web	suporte existente limitado, ou também por meio de ferramentas de terceiros como o ViewVC	apenas por meio de ferramentas de terceiros como o ViewVC	apenas por meio de ferramentas de terceiros como o ViewVC
Velocidade	um pouco mais lento	um pouco mais rápido	um pouco mais rápido
Configuração inicial	um tanto complexa	extremamente simples	moderadamente simples

Escolhendo uma Configuração de Servidor

Então, que servidor você deve usar? Qual é melhor?

Obviamente, não há uma resposta definitiva para esta pergunta. Cada equipe tem diferentes necessidades e os diferentes servidores todos representam diferentes conjuntos de características. O projeto Subversion em si não endossa um ou outro servidor, nem mesmo considera um servidor mais "oficial" que outro.

Aqui estão algumas razões pelas quais você deveria escolher uma configuração ao invés de outra, bem como as razões pelas quais você *não* deveria escolher uma delas.

O Servidor svnserve

Porque você pode querer usá-lo:

- Rápido e fácil de configurar.
- Protocolo de rede orientado a estado e notavelmente mais rápido que o WebDAV.
- Dispensa necessidade da criação de contas de usuário no sistema servidor.
- Senhas não trafegam através da rede.

Porque você pode querer evitá-lo:

- Protocolo de rede não é criptografado.

- Apenas um único método de autenticação disponível.
- Senhas são armazenadas em texto puro no servidor.
- Sem nenhum tipo de log, mesmo para erros.

svnserve sobre SSH

Porque você pode querer usá-lo:

- Protocolo de rede orientado a estado e notavelmente mais rápido que o WebDAV.
- Você pode aproveitar a existência de contas ssh e infraestrutura de usuários existente.
- Todo o tráfego de rede é criptografado.

Porque você pode querer evitá-lo:

- Apenas um único método de autenticação disponível.
- Sem nenhum tipo de log, mesmo para erros.
- Necessita que os usuários estejam num mesmo grupo no sistema, ou que usem uma chave ssh compartilhada.
- Seu uso inadequado pode resultar em problemas com permissões de arquivos.

O Servidor Apache HTTP

Porque você pode querer usá-lo:

- Permite que o Subversion use quaisquer dos inúmeros sistemas de autenticação já disponíveis e integrados com o Apache.
- Dispensa necessidade de criação de contas de usuário no sistema servidor.
- Logs completos do Apache.
- Tráfego de rede pode ser criptografado com SSL.
- HTTP(S) quase sempre não tem problemas para passar por firewalls.
- Navegação no repositório através de um navegador web.
- Repositório pode ser montado como um drive de rede para controle de versão transparente. (Veja “Autoversionamento”.)

Porque você pode querer evitá-lo:

- Notavelmente mais lento que o svnserve, pelo fato do HTTP ser um protocolo sem informação de estado e acabar demandando mais requisições.
- Configuração inicial pode ser complexa.

Recomendações

No geral, os autores deste livro recomendam uma instalação tradicional do **svnserve** para pequenas equipes que ainda estão tentando familiarizar-se com o servidor Subversion; é a forma mais simples de utilização, e a que demanda menos esforço de manutenção. Você sempre pode trocar para uma implantação de servidor mais complexa conforme suas necessidades mudem.

Aqui seguem algumas recomendações e dicas em geral, baseadas na experiência de vários anos de suporte a usuários:

- Se você está tentando configurar o servidor mais simples possível para seu grupo, então uma instalação tradicional do **svnserve** é o caminho mais fácil e rápido. Note, entretanto, que os dados de seu repositório vão ser transmitidos às claras pela rede. Se você estiver fazendo uma implantação inteiramente dentro de sua rede LAN ou VPN da sua empresa, isto não chega a ser nenhum problema. Mas se o repositório tiver de ser acessível pela internet, então você deveria se assegurar que o conteúdo do repositório não contém dados sensíveis (p.ex. se é apenas código-fonte.)

- Se você precisar de integração com alguns sistemas de identificação existentes (LDAP, Active Directory, NTLM, X.509, etc.), então uma configuração baseada no Apache será sua única opção. Similarmente, se você indispensavelmente precisar de log de servidor tanto para registro de erros de servidor quanto para atividades dos clientes, então um servidor com base no Apache será necessário.

- Se você optou por usar ou o Apache ou o **svnserve**, crie uma única conta no sistema para o usuário `svn` e faça com que o processo do servidor seja executado por este usuário. Certifique-se de fazer com que o diretório do repositório pertença totalmente ao usuário `svn` também. Do ponto de vista da segurança, isto deixa os dados do repositório adequadamente seguros e protegidos pelas permissões do sistema de arquivos do sistema operacional, alteráveis apenas pelo próprio processo do servidor Subversion.

- Se você já tiver uma infraestrutura fortemente baseada em contas SSH, e se seus usuários já possuírem contas no servidor, então faz sentido implantar uma solução usando o svnserve sobre SSH. Caso contrário, não recomendamos esta opção largamente ao público. Ter seus usuários acessando o repositório por meio de contas (imaginárias) gerenciadas pelo **svnserve** ou pelo Apache é geralmente considerado mais seguro do que ter acesso por meio de contas reais no sistema. Se o motivo para você fazer isso for apenas obter uma comunicação criptografada, nós recomendamos que você utilize Apache com SSL no lugar.

- *Não* e entusiasme simplesmente com a idéia de ter todos os seus usuários acessando o repositório diretamente através de URLs `file://`. Mesmo se o repositório estiver disponível a todos para leitura através de um compartilhamento de rede, isto é uma má idéia. Esta configuração remove quaisquer camadas de proteção entre os usuários e o repositório: os usuários podem acidentalmente (ou intencionalmente) corromper a base de dados do repositório, dificulta deixar o repositório offline para fins de inspeção ou atualização, e ainda podem surgir problemas relacionados a permissões de arquivos (veja “Dando Suporte a Múltiplos Métodos de Acesso ao Repositório”.) Perceba que esta é uma das razões pelas quais nós alertamos acerca do acesso aos repositórios através de URLs `svn+ssh://`—o que sob a perspectiva de segurança, é efetivamente o mesmo que ter usuários locais acessando via `file://`, e pode resultar nos mesmos problemas se o administrador não for cuidadoso.

svnserve, um servidor especializado

O programa **svnserve** é um servidor leve, capaz de falar com clientes via TCP/IP usando um protocolo específico e robusto. Os clientes contactam um servidor **svnserve** usando URLs que começam com o esquema `svn://` ou `svn+ssh://`. Esta seção vai explicar as diversas formas de se executar o **svnserve**, como os clientes se autenticam para o servidor, e como configurar o controle de acesso apropriado aos seus repositórios.

Invocando o Servidor

Há poucas maneiras distintas de se invocar o programa **svnserve**:

- Executar o **svnserve** como um daemon independente, aguardando por requisições.

- Fazer com que o daemon **inetd** do Unix dispare temporariamente o **svnserve** a cada vez que uma requisição chegar numa dada porta.

- Fazer com que o SSH execute um **svnserve** temporário sobre um túnel criptografado.

- Executar o **svnserve** como um serviço do Windows.

svnserve como Daemon

A opção mais fácil é executar o **svnserve** como um “daemon” independente. Use a opção `-d` para isto:

```
$ svnserve -d
$               # o svnserve está rodando agora, ouvindo na porta 3690
```

Ao executar o **svnserve** no modo daemon, você pode usar as opções `--listen-port=` e `--listen-host=` para especificar a porta e o hostname exatos aos quais o servidor estará “associado”.

Uma vez que tenhamos iniciado o **svnserve** como mostrado acima, isto torna todos os repositórios do sistema disponíveis na rede. Um cliente precisa especificar uma URL com um caminho *absoluto* do repositório. Por exemplo, se um repositório estiver localizado em `/usr/local/repositories/project1`, então um cliente deveria acessá-lo com `svn://host.example.com/usr/local/repositories/project1`. Para aumentar a segurança, você pode passar a opção `-r` para o **svnserve**, o que limita a exportar apenas os repositórios sob o caminho especificado. Por exemplo:

```
$ svnserve -d -r /usr/local/repositories
…
```

O uso da opção `-r` efetivamente modifica o local que o programa considera como a raiz do sistema de arquivos remoto. Os clientes então usam URLs com aquela parte do caminho removida, tornando-as mais curtas (e bem menos informativas):

```
$ svn checkout svn://host.example.com/project1
…
```

svnserve através do inetd

Se você quiser que o **inetd** execute o processo, então você precisa passar a opção `-i` (`--inetd`). No exemplo, mostramos a saída da execução do comando `svnserve -i` na linha de comando, mas note que atualmente não é assim que se inicia o daemon; leia os parágrafos depois do exemplo para saber como configurar o **inetd** para iniciar o **svnserve**.

```
$ svnserve -i
( success ( 1 2 ( ANONYMOUS ) ( edit-pipeline ) ) )
```

Quando invocado com a opção `--inetd`, o **svnserve** tenta se comunicar com um cliente Subversion por meio do *stdin* e *stdout* usando um protocolo específico. Este é o comportamento padrão para um programa sendo executado através do **inetd**. A IANA reservou a porta 3690 para o protocolo Subversion, assim, em um sistema Unix-like você poderia adicionar linhas como estas ao arquivo `/etc/services` (se elas já não existirem):

```
svn           3690/tcp   # Subversion
svn           3690/udp   # Subversion
```

E se seu sistema Unix-like estiver usando um daemon **inetd** clássico, você pode adicionar esta linha ao arquivo `/etc/inetd.conf`:

```
svn stream tcp nowait svnowner /usr/bin/svnserve svnserve -i
```

Assegure-se de que “svnowner” seja um usuário com permissões apropriadas para acesso aos seus repositórios. Agora, quando uma conexão do cliente atingir seu servidor na porta 3690, o **inetd** vai disparar um processo **svnserve** para atendê-la. Obviamente, você também pode querer adicionar a opção `-r` para restringir quais repositórios serão exportados.

svnserve sobre um Túnel

Uma terceira forma de se invocar o **svnserve** é no "modo túnel", com a opção `-t`. Este modo assume que um programa de acesso remoto como o **RSH** ou o **SSH** autenticou um usuário com sucesso e está agora invocando um processo **svnserve** particular *como aquele usuário*. (Note que você, o usuário, vai raramente, ou talvez nunca, precisar invocar o **svnserve** com a opção `-t` na linha de comando; já que o próprio daemon **SSH** faz isso para você.) O programa **svnserve** funciona normalmente (se comunicando por meio do *stdin* e *stdout*), e assume que o tráfego está sendo automaticamente redirecionado por algum tipo de túnel de volta para o cliente. Quando o **svnserve** é invocado por um túnel agente como este, assegure-se de que o usuário autenticado tenha completo acesso de leitura e escrita aos arquivos da base de dados do repositório. É essencialmente o mesmo que um usuário local acessando o repositório por meio de URLs `file://`.

Esta opção está descrita com mais detalhes em "Tunelamento sobre SSH".

svnserve como um Serviço do Windows

Se seu sistema Windows é descendente dos Windows NT (2000, 2003, XP, Vista), então você pode executar o **svnserve** como um serviço padrão do Windows. Esta é tipicamente uma experiência mais proveitosa do que executá-lo como um daemon independente com a opção `--daemon (-d)`. Usar o modo daemon implica em executar um console, digitar um comando, e então deixar a janela do console executando indefinidamente. Um serviço do Windows, no entanto, executa em segundo plano, pode ser executado automaticamente na inicialização, e pode se iniciado e parado através da mesma interface de administração como os outros serviços do Windows.

Você vai precisar definir o novo serviço usando a ferramenta de linha de comando **SC.EXE.**. Semelhantemente à linha de configuração do **inetd**, você deve especificar a forma exata de invocação do **svnserve** para que o Windows o execute na inicialização:

```
C:\> sc create svn
        binpath= "C:\svn\bin\svnserve.exe --service -r C:\repos"
        displayname= "Servidor Subversion"
        depend= Tcpip
        start= auto
```

Isto define um novo serviço do Windows chamado "svn", o qual executa um comando **svnserve.exe** particular quando iniciado (neste caso, com raiz em `C:\repos`.) No entanto, há diversos pontos a considerar neste exemplo anterior.

Primeiramente, note que o programa **svnserve.exe** deve sempre ser chamado com a opção `--service`. Quaisquer outras opções para o **svnserve** então devem ser especificadas na mesma linha, mas você não pode adicionar opções conflitantes tais como `--daemon (-d)`, `--tunnel`, ou `--inetd (-i)`. Já opções como `-r` ou `--listen-port` não terão problemas. Em segundo lugar, tenha cuidado com relação a espaços ao invocar o comando **SC.EXE**: padrões `chave= valor` não devem conter espaços entre `chave=` e devem ter exatamente um espaço antes de `valor`. Por último, tenha cuidado também com espaços na sua linha de comando a ser executada. Se um nome de diretório contiver espaços (ou outros caracteres que precisem de escape), coloque todo o valor interno de `binpath` entre aspas duplas, escapando-as:

```
C:\> sc create svn
        binpath= "\"C:\arquivos de programas\svn\bin\svnserve.exe\" --service -r C:\repo
        displayname= "Servidor Subversion"
        depend= Tcpip
        start= auto
```

Também observe que o termo `binpath` é confuso—seu valor é a *linha de comando*, não o caminho para um executável. Por isso que você precisa delimitá-lo com aspas se o valor contiver espaços.

Uma vez que o serviço esteja definido, ele pode ser parado, iniciado, ou consultado usando-se as ferramentas GUI (o painel de controle administrativo Serviços), bem como através da linha de comando:

```
C:\> net stop svn
C:\> net start svn
```

O serviço também pode ser desinstalado (i.e. indefinido) excluindo-se sua definição: `sc delete svn`. Apenas certifique-se de parar o serviço antes! O programa **SC.EXE** tem diversos outros subcomandos e opções; digite `sc /?` para saber mais sobre ele.

Autenticação e autorização internos

Quando um cliente se conecta a um processo **svnserve**, as seguintes coisas acontecem:

- O cliente seleciona um repositório específico.
- O servidor processa o arquivo `conf/svnserve.conf` do repositório e começa a tomar medidas para ratificar quaisquer políticas de autenticação e autorização nele definidas.
- Dependendo da situação e das políticas de autorização,
  - ao cliente pode ser permitido fazer requisições de forma anônima, sem mesmo precisar receber um desafio de autenticação, OU
  - o cliente pode ser desafiado para se autenticar a qualquer tempo, OU
  - se operando em “modo túnel”, o cliente irá declarar a si próprio como já tendo sido externamente autenticado.

Até o momento em que este livro estava sendo escrito, o servidor sabia apenas como enviar desafios de autenticação do tipo CRAM-MD5 [1] Essencialmente, o servidor envia uma pequena quantidade de dados para o cliente. O cliente usa o algoritmo de hash MD5 para criar uma impressão digital dos dados e da senha combinados, então envia esta impressão digital como resposta. O servidor realiza a mesma computação com a senha armazenada para verificar se os resultados coincidem. *Em nenhum momento a senha atual é trafegada pela rede.*

E claro, também é possível para o cliente ser autenticado externamente por meio de um túnel agente, tal como o **SSH**. Neste caso, o servidor simplesmente examina o usuário com o qual está sendo executado, e o utiliza como nome de usuário autenticado. Para mais detalhes sobre isto, veja “Tunelamento sobre SSH”.

Como você já deve ter percebido, o arquivo `svnserve.conf` do repositório é o mecanismo central para controle das políticas de autenticação e autorização. O arquivo possui o mesmo formato que outros arquivos de configuração (veja “Área de Configuração do Tempo de Execução”): nomes de seção são marcados por colchetes (`[` e `[`), comentários iniciam por cerquilha (#), e cada seção contém variáveis específicas que podem ser definidas (`variável = valor`). Vamos conferir estes arquivos e aprender como usá-los.

Criar um arquivo 'users' e um domínio

Por hora, a seção `[general]` do `svnserve.conf` tem todas as variáveis que você precisa. Comece alterando os valores dessas variáveis: escolha um nome para um arquivo que irá conter seus nomes de usuários e senha, e escolha um domínio de autenticação:

```
[general]
password-db = userfile
realm = example domain
```

[1] Consulte a RFC 2195.

O domínio (`realm`) é um nome que você define. Ele informa aos clientes a que tipo de “espaço de nomes de autenticação” você está se conectando; o cliente Subversion o exibe no prompt de autenticação, e o utiliza como chave (junto com o hostname do servidor e a porta) para fazer cache de credenciais no disco (veja “Armazenando Credenciais no Cliente”). A variável `password-db` aponta para um arquivo em separado que contém uma lista de nomes de usuários e senhas, usando o mesmo formato familiar. Por exemplo:

```
[users]
harry = foopassword
sally = barpassword
```

O valor de `password-db` pode ser um caminho absoluto ou relativo para o arquivo de usuários. Para muitos administradores, é fácil manter o arquivo logo dentro da área `conf/` do repositório, juntamente com o `svnserve.conf`. Por outro lado, é possível que você queira ter dois ou mais repositórios compartilhando o mesmo arquivo de usuários; neste caso, o arquivo provavelmente deve ficar em um local mais público. Repositórios que compartilhem o arquivo de usuários também devem ser configurados para ter um mesmo domínio, uma vez que a lista de usuários essencialmente define um domínio de autenticação. Onde quer que este arquivo esteja, certifique-se de definir as permissões de leitura e escrita adequadamente. Se você sabe com qual(is) usuário(s) o **svnserve** irá rodar, restrinja o acesso de leitura ao arquivo de usuários conforme necessário.

Definindo controles de acesso

Há ainda mais duas variáveis para definir no arquivo `svnserve.conf`: elas determinam o que os usuários não autenticados (anônimos) e os usuários autenticados têm permissão de fazer. As variáveis `anon-access` e `auth-access` podem ser definidas para os valores `none`, `read`, ou `write`. Atribuindo o valor `none` você proíbe tanto a leitura quanto a escrita; com `read` você permite acesso somente leitura ao repositório, enquanto que `write` permite acesso completo de leitura/escrita ao repositório. Por exemplo:

```
[general]
password-db = userfile
realm = example domain

# usuários anônimos pode apenas ler o repositório
anon-access = read

# usuários autenticados podem tanto ler quanto escrever
auth-access = write
```

De fato, as configurações deste exemplo são os valores padrão para as variáveis, você poderia esquecer de definí-las. Se você quer ser ainda mais conservador, você pode bloquear o acesso anônimo completamente:

```
[general]
password-db = userfile
realm = example realm

# usuários anônimos não são permitidos
anon-access = none

# usuários autenticados podem tanto ler quanto escrever
auth-access = write
```

O processo servidor não entende apenas esta “restrição” no controle de acesso ao repositório, mas também restrições de acesso mais granularizadas definidas para arquivos ou diretórios específicos

dentro do repositório. Para usar este recurso, você precisa criar um arquivo contendo regras mais detalhadas, e então definir o valor da variável `authz-db` para o caminho que o aponte:

```
[general]
password-db = userfile
realm = example realm

# Regras de acesso para locais específicos
authz-db = authzfile
```

A sintaxe do arquivo `authzfile` é discutida em mais detalhes em "Autorização Baseada em Caminhos". Atente que a variável `authz-db` não é mutuamente exclusiva com as variáveis `anon-access` e `auth-access`; se todas elas estiverem definidas ao mesmo tempo, então *todas* as regras devem ser satisfeitas antes que o acesso seja permitido.

Tunelamento sobre SSH

A autenticação interna do **svnserve** pode ser bastante útil, pois evita a necessidade de se criar contas reais no sistema. Por outro lado, alguns administradores já possuem frameworks de autenticação com SSH bem estabelecidos em funcionamento. Nestas situações, todos os usuários do projeto devem já ter contas no sistema e a possibilidade "dar um SSH" para acessar a máquina servidora.

É fácil usar o SSH juntamente com o **svnserve**. O cliente simplesmente usa o esquema `svn+ssh://` na URL para conectar:

```
$ whoami
harry

$ svn list svn+ssh://host.example.com/repos/project
harry@host.example.com's password:  *****

foo
bar
baz
…
```

Neste exemplo, o cliente Subversion está invocando um processo **ssh** local, conectando-se a `host.example.com`, autenticando-se como usuário `harry`, então disparando um processo **svnserve** particular na máquina remota rodando como o usuário `harry` O comando **svnserve** está sendo invocado através em modo túnel (`-t`) e seu protocolo de rede está sendo "tunelado" pela conexão criptografada pelo **ssh**, o túnel agente. O **svnserve** sabe que está sendo executando pelo usuário `harry`, e se o cliente executar um commit, o nome do usuário autenticado será usado como autor da nova revisão.

A coisa importante a compreender aqui é que o cliente Subversion *não* está se conectando a um daemon **svnserve** em execução. Este método de acesso não requer um daemon, nem tampouco percebe se há algum daemon presente. Este método se baseia totalmente na capacidade do **ssh** de executar um processo **svnserve** temporário, que então termina quando a conexão de rede é fechada.

Ao usar URLs `svn+ssh://` para acessar um repositório, lembre-se que é o programa **ssh** que está solicitando autenticação, e *não* o programa cliente **svn**. Isto quer dizer que aqui não há cache automático de senhas acontecendo (veja "Armazenando Credenciais no Cliente"). O cliente Subversion quase sempre faz múltiplas conexões ao repositório, apesar de que os usuários normalmente não percebem isto devido a este recurso de cache de senhas. Ao usar URLs `svn+ssh://`, entretanto, os usuários podem ser incomodados repetidamente pelo **ssh** solicitando senhas a cada conexão que inicie. A solução é usar uma ferramenta separada para cache de senhas do SSH como o **ssh-agent** em um sistema Unix-like, ou o **pageant** no Windows.

Quando executada sobre um túnel, a autorização é principalmente controlada pelas permissões do sistema operacional para os arquivos da base dados do repositório; o que é praticamente o mesmo como se Harry estivesse acessando o repositório diretamente através de uma URL `file://`. Se múltiplos usuários no sistema vão acessar o repositório diretamente, você pode querer colocá-los num mesmo grupo, e você precisará ter cuidado com as umasks. (Não deixe de ler "Dando Suporte a Múltiplos Métodos de Acesso ao Repositório".) Mas mesmo no caso do tunelamento, o arquivo `svnserve.conf` ainda pode ser usado para bloquear acesso, simplesmente definindo `auth-access = read` ou `auth-access = none`. [2]

Você pode ter pensado que essa história de tunelamento com SSH acabou por aqui, mas não. O Subversion permite que você crie comportamentos específicos para o modo túnel em seu arquivo `config` de execução (veja "Área de Configuração do Tempo de Execução"). Por exemplo, suponha que você queira usar o RSH ao invés do SSH[3]. Na seção `[tunnels]` do seu arquivo `config`, simplesmente defina algo parecido com isto:

```
[tunnels]
rsh = rsh
```

E agora, você pode usar este nova definição de túnel usando um esquema de URL que casa com o nome de sua nova variável: `svn+rsh://host/path`. Ao usar o novo esquema de URL, o cliente Subversion atualmente vai ser executado pelo comando **rsh host svnserve -t** por trás dos panos. Se você incluir um nome de usuário na URL (por exemplo, `svn+rsh://username@host/path`) o cliente também vai incluí-lo em seu comando (**rsh username@host svnserve -t**). Mas você pode definir novos esquemas de tunelamento que sejam muito mais inteligentes que isto:

```
[tunnels]
joessh = $JOESSH /opt/alternate/ssh -p 29934
```

Este exemplo demonstra uma porção de coisas. Primeiro, ele mostra como fazer o cliente do Subversion executar um binário de tunelamento bem específico (este que está localizado em `/opt/alternate/ssh`) com opções específicas. Neste caso, acessando uma URL `svn+joessh://` deveria invocar o binário SSH em questão com `-p 29934` como argumentos—útil se você quer que o programa do túnel se conecte a uma porta não-padrão.

Segundo, esse exemplo mostra como definir uma variável de ambiente personalizada que pode sobrescrever o nome do programa de tunelamento. Definir a variável de ambiente `SVN_SSH` é uma maneira conveniente de sobrescrever o túnel agente SSH padrão. Mas se você precisar fazer sobrescrita diversas vezes para diferentes servidores, cada um talvez contactando uma porta diferente ou passando diferentes conjuntos de opções para o SSH, você pode usar o mecanismo demonstrado neste exemplo. Agora se formos definir a variável de ambiente `JOESSH`, seu valor irá sobrescrever o valor inteiro da variável túnel—**$JOESSH** deverá ser executado ao invés de **/opt/alternate/ssh -p 29934**.

Dicas de configuração do SSH

Não é possível apenas controlar a forma como o cliente invoca o **ssh**, mas também controlar o comportamento do **ssh** em sua máquina servidora. Nesta seção, vamos mostrar como controlar exatamente o comando **svnserve** executado pelo **sshd**, além de como ter múltiplos usuários compartilhando uma única conta no sistema.

Configuração inicial

Para começar, localize o diretório home da conta que você vai usar para executar o **svnserve**. Certifique-se de que a conta tenha um par de chaves pública/privada instalado, e que o usuário consiga

[2]Perceba que usar qualquer tipo de controle de acesso através do **svnserve** acaba não fazendo muito sentido; o usuário sempre possui acesso direto à base de dados do repositório.
[3]Atualmente nós não recomendamos isto, uma vez que o RSH é sabidamente menos seguro que o SSH.

ter acesso ao sistema por meio de autenticação com chave pública. A autenticação por senha não irá funcionar, já que todas as seguintes dicas sobre SSH estão relacionadas com o uso do arquivo `authorized_keys` do SSH.

Se ainda não existir, crie o arquivo `authorized_keys` (no Unix, tipicamente em `~/.ssh/authorized_keys`). Cada linha neste arquivo descreve uma chave pública com permissão para conectar. As linhas são comumente da forma:

```
ssh-dsa AAAABtce9euch… user@example.com
```

O primeiro campo descreve o tipo da chave, o segundo campo é a chave em si codificada em base64, e o terceiro campo é um comentário. Porém, é pouco conhecido o fato de que a linha inteira pode ser precedida por um campo `command`:

```
command="program" ssh-dsa AAAABtce9euch… user@example.com
```

Quando o campo `command` é definido, o daemon SSH irá rodar o programa descrito em vez da típica invocação **svnserve -t** que o cliente Subversion está esperando. Isto abre espaço para um conjunto de truques no lado do servidor. Nos exemplos a seguir, abreviamos a linha do arquivo para:

```
command="program" TIPO CHAVE COMENTÁRIO
```

Controlando o comando invocado

Pelo fato de podermos especificar o comando executado no lado do servidor, é fácil determinar um binário **svnserve** específico para rodar e para o qual passar argumentos extras:

```
command="/caminho/do/svnserve -t -r /virtual/root" TIPO CHAVE
COMENTÁRIO
```

Neste exemplo, `/caminho/do/svnserve` pode ser um script específico que encapsule uma chamada ao **svnserve** e que configure o umask (veja “Dando Suporte a Múltiplos Métodos de Acesso ao Repositório”). O exemplo também mostra como prender o **svnserve** em um diretório raiz virtual, tal como sempre ocorre ao se executar o **svnserve** como um processo daemon. Isto pode ser feito tanto para restringir o acesso a partes do sistema, ou simplesmente para facilitar para que o usuário não tenha de digitar um caminho absoluto na URL `svn+ssh://`.

Também é possível ter múltiplos usuários compartilhando uma única conta. Ao invés de criar uma conta em separado no sistema para cada usuário, gere um par de chaves pública/privada para cada pessoa. Então ponha cada chave pública dentro do arquivo `authorized_keys`, uma por linha, e utilize a opção `--tunel-user`:

```
command="svnserve -t --tunnel-user=harry" TIPO1 CHAVE1 harry@example.com
command="svnserve -t --tunnel-user=sally" TIPO2 CHAVE2 sally@example.com
```

Este exemplo permite que Harry e Sally se conectem à mesma conta por meio da autenticação de chave pública. Cada um deles tem um comando específico a ser executado; a opção `--tunnel-user` diz ao **svnserve -t** para assumir que o argumento informado é um nome de usuário autenticado. Não fosse pelo `--tunnel-user` pareceria como se todos os commits viessem de uma única mesma conta de sistema compartilhada.

Uma última palavra de precaução: dando acesso ao usuário através de uma chave pública numa conta compartilhada deve permitir ainda outras formas de acesso por SSH, ainda que você já tenha definido o valor do campo `command` no `authorized_keys`. Por exemplo, o usuário pode ainda ter acesso a um shell através do SSH, ou ser capaz de executar o X11 ou outro tipo de redirecionamento de portas

através do servidor. Para dar ao usuário o mínimo de permissão possível, você pode querer especificar algumas opções restritivas imediatamente após o `command`:

```
command="svnserve -t --tunnel-user=harry",no-port-forwarding,\
        no-agent-forwarding,no-X11-forwarding,no-pty \
        TIPO1 CHAVE1 harry@example.com
```

httpd, o servidor HTTP Apache

O servidor Apache HTTP é um servidor de rede "robusto" do qual o Subversion pode tirar proveito. Por meio de um módulo específico, o **httpd** torna os repositórios Subversion disponíveis aos clientes por meio do protocolo WebDAV/DeltaV, que é uma extensão ao HTTP 1.1 (consulte http://www.webdav.org/ para mais informações). Este protocolo usa o onipresente protocolo HTTP, que é o coração da World Wide Web, e adiciona capacidades de escrita—especificamente, escrita sob controle de versão. O resultado é um sistema robusto, padronizado e que está convenientemente empacotado como uma parte do software Apache 2.0, é suportado por vários sistemas operacionais e produtos de terceiros, e não requer administradores de rede para abrir nenhuma outra porta específica. [4] Apesar de o servidor Apache-Subversion ter mais recursos que o **svnserve**, ele também é um pouco mais difícil de configurar. Um pouco mais de complexidade é o preço a se pagar por um pouco mais de flexibilidade.

Muitas das discussões a seguir incluem referências a diretivas de configuração do Apache. Por mais que sejam dados exemplos de uso destas diretivas, descrevê-las por completo está fora do escopo deste capítulo. A equipe do Apache mantém excelente documentação, disponível publicamente no seu website em http://httpd.apache.org. Por exemplo, uma referência geral das diretivas de configuração está localizada em http://httpd.apache.org/docs-2.0/mod/directives.html.

Além disso, quando você faz alteração na configuração do seu Apache, é bem provável que erros sejam cometidos em algum ponto. Se você ainda não está ambientado com o subsistema de logs do Apache, você deveria familiarizar-se com ele. Em seu arquivo `httpd.conf` estão as diretivas que especificam os caminhos em disco dos logs de acesso e de erros gerados pelo Apache (as diretivas `CustomLog` e `ErrorLog`, respectivamente). O módulo mod_dav_svn do Subversion também usa a interface de logs de erro do Apache. A qualquer momento você pode verificar o conteúdo destes arquivos, buscando informação que pode revelar a causa de um problema que não seria claramente percebido de outra forma.

Por que o Apache 2?

Se você é um administrador de sistemas, é bem provável que você já esteja rodando o servidor web Apache e tenha alguma experiência anterior com ele. No momento em que este livro era escrito, o Apache 1.3 é de longe a versão mais popular do Apache. O mundo tem sido um tanto lento para atualizar para o Apache da série 2.X por várias razões: algumas pessoas têm medo da mudança, especialmente mudança em algo tão crítico como um servidor web. Outras pessoas dependem de módulos de plug-in que só funcionam sobre a API do Apache 1.3, e estão aguardando que estes sejam portados para a 2.X. Qualquer que seja a razão, muitas pessoas começam a se preocupar quando descobrem que o módulo Apache do Subversion é escrito especificamente para a API do Apache 2.

A resposta adequada a este problema é: não se preocupe com isto. É fácil executar o Apache 1.3 e o Apache 2 lado a lado; simplesmente instale-os em locais separados, e use o Apache 2 como um servidor Subversion dedicado que execute em outra porta que não a 80. Os clientes podem acessar o repositório inserindo o número da porta na URL:

```
$ svn checkout http://host.example.com:7382/repos/project
…
```

[4] Eles realmente detestam fazer isso.

Pré-requisitos

Para disponibilizar seus repositórios em rede via HTTP, você basicamente necessita de quatro componentes, disponíveis em dois pacotes. Você vai precisar do Apache **httpd** 2.0, do módulo DAV **mod_dav** que já vem com ele, do Subversion, e do **mod_dav_svn**, módulo que provê acesso ao sistema de arquivos distribuído junto com o Subversion. Uma vez que você tenha todos estes componentes, o processo de disponibilizar seus repositórios em rede é tão simples quanto:

- executar o httpd 2.0 com o módulo mod_dav,
- instalar o plugin mod_dav_svn no mod_dav, que utiliza as bibliotecas do Subversion para acessar o repositório e
- configurar seu arquivo `httpd.conf` para exportar (ou expor) o repositório.

Você pode realizar os primeiros dois passos tanto compilando o **httpd** e o Subversion a partir do código-fonte, ou instalando os pacotes binários pré-construídos para o seu sistema. Para informações mais atualizadas sobre como compilar o Subversion para uso com o servidor Apache HTTP, além de como compilar e configurar o próprio Apache para este propósito, veja o arquivo `INSTALL` na pasta de mais alto nível na árvore do código-fonte do Subversion.

Configuração Básica do Apache

Tendo instalado todos os componentes necessários em seu sistema, tudo o que resta é a configuração do Apache por meio de seu arquivo `httpd.conf`. Indique ao Apache para carregar o módulo mod_dav_svn usando a diretiva `LoadModule`. Esta diretiva deve preceder a qualquer outro item de configuração relacionado ao Subversion. Se seu Apache foi instalado usando sua estrutura padrão, seu módulo **mod_dav_svn** deve estar instalado no subdiretório `modules` do local de instalação do Apache (comumente em `/usr/local/apache2`). A diretiva `LoadModule` tem uma sintaxe simples, mapeando um nome de módulo ao local em disco da correspondente biblioteca compartilhada:

```
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

Note que se o **mod_dav** foi compilado como um objeto compartilhado (ao invés de ter sido lincado diretamente ao binário **httpd**), você vai precisar de uma declaração `LoadModule` para ele, também. Assegure-se de que sua declaração venha antes da linha **mod_dav_svn**:

```
LoadModule dav_module         modules/mod_dav.so
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

Em outra parta de seu arquivo de configuração, você agora vai dizer ao Apache onde você mantém seu repositório (ou repositórios) Subversion. A diretiva `Location` tem uma notação parecida com a de XML, começando com uma tag de abertura, e terminando com uma tag de fechamento, com várias outras diretivas de configuração no meio. O propósito da diretiva `Location` é instruir o Apache a fazer algo especial quando manipular requisições que sejam direcionadas a uma certa URL ou uma de suas filhas. No caso do Subversion, quer que o Apache simplesmente desconsidere URLs que apontem para recursos sob controle de versão na camada DAV. Você pode instruir o Apache a delegar a manipulação de todas as URL em que cuja parte do caminho (a parte da URL após o nome do servidor e do número de porta opcional) comece com `/repos/` para um provedor DAV cujo repositório está localizado em `/caminho/absoluto/do/repositorio` usando a seguinte sintaxe do `httpd.conf`:

```
<Location /repos>
  DAV svn
  SVNPath /caminho/absoluto/do/repositório
```

```
</Location>
```

Se você pretende disponibilizar múltiplos repositórios Subversion que se encontrem sob um mesmo diretório-pai em seu disco local, você pode usar uma diretiva alternativa, a diretiva `SVNParentPath`, para indicar este diretório-pai comum. Por exemplo, se você sabe que você vai criar múltiplos repositórios Subversion num diretório `/usr/local/svn` que poderia ser acessado a partir de URLs como `http://my.server.com/svn/repos1`, `http://my.server.com/svn/repos2`, e assim por diante, você pode usar a sintaxe de configuração do `httpd.conf` do exemplo a seguir:

```
<Location /svn>
  DAV svn

  # qualquer URL "/svn/foo" vai mapear para um repositório /usr/local/svn/foo
  SVNParentPath /usr/local/svn
</Location>
```

Usando a sintaxe anterior, o Apache vai delegar a manipulação de todas as URLs cuja a parte do caminho comece com `/svn/` para o provedor DAV do Subversion, que então vai assumir que quaisquer itens no diretório especificado pela diretiva `SVNParentPath` atualmente são repositórios Subversion. Esta é uma sintaxe particularmente conveniente para isto, ao contrário do uso da diretiva `SVNPath`, você não tem que reiniciar o Apache para criar e acessar via rede os novos repositórios.

Certifique-se de que ao definir seu novo `Location`, este não se sobreponha a outros Locations exportados. Por exemplo, se seu `DocumentRoot` principal estiver exportado para `/www`, não exporta um repositório Subversion em `<Location /www/repos>`. Se vier uma requisição para a URL `/www/repos/foo.c`, o Apache não vai saber se deve procurar pelo arquivo `repos/foo.c` no `DocumentRoot`, ou se delega ao **mod_dav_svn** para que este retorne `foo.c` a partir do repositório Subversion. O resultado quase sempre é um erro de servidor no formato `301 Moved Permanently`.

Nomes de Servidores e a Requisição COPY

Subversion faz uso da requisição do tipo `COPY` para executar cópias de arquivos e diretórios do lado do servidor. Como parte da verificação de integridade é feita pelos módulos do Apache, espera-se que a origem da cópia esteja localizada na mesma máquina que o destino da cópia. Para satisfazer este requisito, você vai precisar dizer ao mod_dav o nome do hostname usado em seu servidor. Geralmente, você pode usar a diretiva `ServerName` no `httpd.conf` para fazer isto.

```
ServerName svn.example.com
```

Se você está usando o suporte a hosts virtuais do Apache através da diretiva `NameVirtualHost`, você pode precisar usar a diretiva `ServerAlias` para especificar nomes adicionais pelos quais seu servidor também é conhecido. Mais uma vez, verifique a documentação do Apache para mais detalhes.

Neste estágio, você deve considerar maciçamente a questão das permissões. Se você já estiver executando o Apache há algum tempo como seu servidor web regular, você provavelmente já tem uma porção de conteúdos—páginas web, scripts e similares. Estes itens já devem ter sido configurados com um conjunto de permissões que lhes possibilitam trabalhar com o Apache, ou mais adequadamente, que permite ao Apache trabalhar com estes arquivos. O Apache, quando usado como um servidor Subversion, também vai precisar das permissões corretas para ler e escrever em seu repositório Subversion.

Você vai precisar determinar uma configuração do sistema de permissões que satisfaça os requisitos do Subversion sem causar problemas em nenhuma página ou script previamente instalado. Isto pode significar alterar as permissões de seu repositório Subversion para corresponder àquelas em uso

por outras coisas que o Apache lhe serve, ou pode significar usar as diretivas `User` e `Group` no `httpd.conf` para especificar que o Apache deve executar como o usuário e grupo de que seja dono do seu repositório Subversion. Não há apenas uma maneira correta de configurar suas permissões, e cada administrador vai ter diferentes razões para fazer as coisas de uma dada maneira. Apenas tenha cuidado pois problemas de permissão são talvez a falha mais comum ao se configurar o repositório Subversion para uso com o Apache.

Opções de Autenticação

Neste ponto, se você configurou o `httpd.conf` para conter algo como

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
</Location>
```

…então seu repositório está “anonimamente” acessível ao mundo. Até que você configure algumas políticas de autenticação e autorização, os repositórios Subversion que você disponibilizar através da diretiva `Location` vão estar globalmente acessíveis a qualquer um. Em outras palavras,

- qualquer pessoa pode usar seu cliente Subversion para dar checkout numa cópia de trabalho de uma URL do repositório (ou qualquer de seus subdiretórios),

- qualquer pessoa pode navegar interativamente pela última revisão do repositório, simplesmente direcionando seu navegador web para a URL do repositório, e

- qualquer pessoa pode dar commit no repositório.

Certamente, você pode já ter definido um hook script para prevenir commits (veja “Implementing Repository Hooks”). Mas conforme você for lendo, verá que também é possível usar os métodos inerentes ao Apache para restringir o acesso de maneiras específicas.

Autenticação HTTP Básica

A maneira mais fácil de autenticar um cliente é através do mecanismo de autenticação HTTP Basic, o qual simplesmente usa um nome de usuário e senha para verificar se o usuário é quem ele diz ser. O Apache provê um utilitário **htpasswd** para gerenciar a lista de nomes de usuários e senhas aceitáveis. Vamos permitir o acesso a commit a Sally e Harry. Primeiro, precisamos adicioná-los ao arquivo de senhas.

```
$ ### Primeira vez: use -c para criar o arquivo
$ ### Use -m para usar criptografia MD5 na senha, o que é mais seguro
$ htpasswd -cm /etc/svn-auth-file harry
New password: *****
Re-type new password: *****
Adding password for user harry
$ htpasswd -m /etc/svn-auth-file sally
New password: *******
Re-type new password: *******
Adding password for user sally
$
```

A seguir, você precisa adicionar mais algumas diretivas do `httpd.conf` dentro de seu bloco `Location` para indicar ao Apache o que fazer com seu novo arquivo de senhas. A diretiva `AuthType` especifica o tipo de sistema de autenticação a usar. Neste caso, vamos especificar o sistema de

autenticação `Basic`. `AuthName` é um nome arbitrário que você dá para o domínio da autenticação. A maioria dos navegadores web vai mostrar este nome da caixa de diálogo quando o navegador estiver perguntando ao usuário por seu nome e senha. Finalmente, use a diretiva `AuthUserFile` para especificar a localização do arquivo de senhas que você criou usando o comando **htpasswd**.

Depois de adicionar estas três diretivas, seu bloco `<Location>` deve ser algo parecido com isto:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /etc/svn-auth-file
</Location>
```

Este bloco `<Location>` ainda não está completo, e não fará nada de útil. Está meramente dizendo ao Apache que sempre que uma autorização for requerida, o Apache deve obter um nome de usuário e senha do cliente Subversion. O que está faltando aqui, entretanto, são diretivas que digam ao Apache *quais* tipos de requisições do cliente necessitam de autorização. Toda vez que uma autorização for requerida, o Apache também irá exigir uma autorização. A coisa mais simples a se fazer é proteger todas as requisições. Adicionar `Require valid-user` indica ao Apache que todas as requisições requerem um usuário autenticado:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /etc/svn-auth-file
  Require valid-user
</Location>
```

Não deixe de ler a próxima seção (“Opções de Autorização”) para mais detalhes sobre a diretiva `Require` e outras formas de definir políticas de autorização.

Uma palavra de alerta: senhas de autenticação HTTP Basic trafegam na rede de forma bem parecida com texto plano, e portanto são extremamente inseguras. Se você está preocupado com a privacidade de suas senhas, pode ser melhor usar algum tipo de criptografia SSL para que o cliente se autentique através de `https://` ao invés de `http://`; no mínimo, você pode configurar o Apache para usar um certificado de servidor auto-assinado. [5] Consulte a documentação do Apache (e a documentação do OpenSSL) sobre como fazê-lo.

Gerência de Certificados SSL

Negócios que precisam expor seus repositórios para acesso por fora do firewall corporativos devem estar conscientes da possibilidade de que pessoas não autorizadas podem “vasculhar” seu tráfego de rede. SSL faz com que esse tipo de análise indesejada tenha menor possibilidade de resultar na exposição de dados sensíveis.

Se um cliente Subversion é compilado para usar OpenSSL, então ele ganha a habilidade de falar com um servidor Apache através de URLs `https://`. A biblioteca Neon usada pelo cliente Subversion não é apenas capaz de verificar certificados de servidor, mas também de prover certificados de cliente quando necessário. Quando cliente e o servidor trocam certificados SSL e se autenticam mutuamente com sucesso, toda a comunicação subseqüente é criptografada por meio de uma chave de sessão.

[5] Por mais que certificados auto-assinados ainda sejam vulneráveis ao “ataque do homem do meio”, tal ataque é muito mais difícil de ser executado por um observador casual, do que o comparado a vasculhar senhas desprotegidas.

Está fora do escopo deste livro descrever como gerar certificados de cliente e de servidor, e como configurar o Apache para usá-los. Muitos outros livros, incluindo a própria documentação do Apache, descrevem esta tarefa. Mas o que *pode* ser coberto aqui é como gerenciar os certificados de cliente e servidor a partir de um cliente Subversion ordinário.

Ao se comunicar com o Apache através de `https://`, um cliente Subversion pode receber dois diferentes tipos de informação:

- um certificado de servidor
- uma demanda por um certificado de cliente

Se o cliente recebe um certificado de servidor, ele precisa verificar se confia no certificado: o servidor é quem ele realmente diz ser? A biblioteca OpenSSL faz isto examinando o assinador do certificado de servidor ou *autoridade certificadora* (AC). Se o OpenSSL for incapaz de confiar automaticamente na AC, ou se algum outro problema ocorrer (como o certificado ter expirado ou o hostname não corresponder), o cliente de linha de comando do Subversion vai lhe perguntar se você quer confiar no certificado do servidor de qualquer maneira:

```
$ svn list https://host.example.com/repos/project

Error validating server certificate for 'https://host.example.com:443':
 - The certificate is not issued by a trusted authority. Use the
   fingerprint to validate the certificate manually!
Certificate information:
 - Hostname: host.example.com
 - Valid: from Jan 30 19:23:56 2004 GMT until Jan 30 19:23:56 2006 GMT
 - Issuer: CA, example.com, Sometown, California, US
 - Fingerprint: 7d:e1:a9:34:33:39:ba:6a:e9:a5:c4:22:98:7b:76:5c:92:a0:9c:7b

(R)eject, accept (t)emporarily or accept (p)ermanently?
```

Este diálogo deve parecer familiar: é essencialmente a mesma pergunta que você provavelmente já tenha visto vindo de seu navegador web (o qual é apenas outro cliente HTTP como o cliente Subversion). Se você escolher a opção (p)ermanente, o certificado do servidor será armazenado em cache em sua área privativa de tempo de execução, `auth/`, da mesma forma que seu nome de usuário e senha são armazenados (veja “Armazenando Credenciais no Cliente”). Uma vez armazenado, o Subversion automaticamente lembrará de confiar neste certificado em negociações futuras.

Seu arquivo em tempo de execução `servers` também lhe possibilita fazer seu cliente Subversion confiar automaticamente em ACs específicas, tanto de forma global quanto discriminadamente por `ssl-authority-files` para uma lista de certificados das ACs com codificação PEM separados por ponto e vírgula:

```
[global]
ssl-authority-files = /caminho/do/CAcert1.pem;/caminho/do/CAcert2.pem
```

Muitas instalações OpenSSL também possuem um conjunto pré-definido de ACs “padrão” que são quase que universalmente confiáveis. Para fazer o cliente Subversion confiar automaticamente nestas autoridades padrão, atribua o valor da variável `ssl-trust-default-ca` para `true`.

Ao conversar com o Apache, o cliente Subversion pode também receber um desafio para um certificado de cliente. O Apache está solicitando que o cliente identifique a si próprio: o cliente é quem ele realmente diz ser? Se tudo prosseguir corretamente, o cliente Subversion manda de volta um certificado privativo assinado por uma AC na qual o Apache confia. O certificado do cliente é comumente armazenado em disco num formato criptografado, protegido por uma senha local. Quando o Subversion recebe este desafio, ele solicitará a você tanto o caminho do certificado quando a senha que o protege:

```
$ svn list https://host.example.com/repos/project

Authentication realm: https://host.example.com:443
Client certificate filename: /caminho/do/meu/cert.p12
Passphrase for '/caminho/do/meu/cert.p12':  ********
…
```

Note que o certificado do cliente é um arquivo no formato “p12”. Para usar um certificado de cliente com o Subversion, este deve estar no formato PKCS#12, que é um padrão portável. A maioria dos navegadores já são capazes de importar e exportar certificados neste formato. Outra opção é usar as ferramentas de linha de comando do OpenSSL para converter certificados existentes para PKCS#12.

Novamente, o arquivo em tempo de execução `servers` permite a você automatizar este desafio numa configuração baseada em host. Qualquer um ou mesmo os dois tipos de informação podem estar descritos em variáveis em tempo de execução:

```
[groups]
examplehost = host.example.com

[examplehost]
ssl-client-cert-file = /caminho/do/meu/cert.p12
ssl-client-cert-password = somepassword
```

Uma vez que você tenha definido as variáveis `ssl-client-cert-file` e `ssl-client-cert-password`, o cliente Subversion pode automaticamente responder a um desafio de certificado de cliente sem solicitar nada a você. [6]

Opções de Autorização

Até este ponto, você configurou a autenticação, mas não a autorização. O Apache é capaz de desafiar cliente e confirmar identidades, mas não está sendo dito como permitir ou restringir acesso a clientes que possuam estas identidades. Esta seção descreve duas estratégias para controle de acesso de seus repositórios.

Controle de Acesso Geral

A maneira mais simples de controle de acesso é autorizar certos usuários ou para acesso somente leitura a um repositório, ou acesso leitura/escrita a um repositório.

The simplest form of access control is to authorize certain users for either read-only access to a repository, or read/write access to a repository.

Você pode restringir acesso em todas as operações de repositório adicionando a diretiva `Require valid-user` ao seu bloco `<Location>`. Usando nosso exemplo anterior, isto significa que apenas aos clientes que afirmaram ser `harry` ou `sally`, e forneceram a senha correta para seus respectivos nomes de usuário será permitido fazer qualquer coisa com o repositório Subversion:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn

  # como autenticar um usuário
```

[6]Pessoas mais preocupadas com segurança podem não querer armazenar a senha do certificado de cliente no arquivo `servers` em tempo de execução.

```
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /caminho/do/arquivo/users

  # apenas usuários autenticados podem acessar o repositório
  Require valid-user
</Location>
```

Algumas vezes você não precisa de regras tão rígidas. Por exemplo, o próprio repositório do código-fonte do Subversion em http://svn.collab.net/repos/svn permite a qualquer um no mundo executar tarefas somente-leitura no repositório (como verificar cópias de trabalho e navegar pelo repositório com um navegador web), mas restringe todas as operações de escrita a usuários autenticados. Para fazer este tipo de restrição seletiva, você pode usar as diretivas de configuração `Limit` e `LimitExcept`. Como a diretiva `Location`, estes blocos possuem tags de abertura e de fechamento, e você deve inserí-las dentro de seu bloco `<Location>`.

Os parâmetros presentes nas diretivas `Limit` e `LimitExcept` são tipos de requisições HTTP que são afetadas por aquele bloco. Por exemplo, se você quiser desabilitar todo o acesso a seu repositório exceto as operações somente-leitura atualmente suportadas, você deve usar a diretiva `LimitExcept`, passando os tipos de requisição `GET`, `PROPFIND`, `OPTIONS`, e `REPORT` como parâmetros. Então a já mencionada diretiva `Require valid-user` deve ser colocada dentro do bloco `<LimitExcept>` ao invés de apenas dentro do bloco `<Location>`.

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn

  # como autenticar um usuário
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /caminho/do/arquivo/users

  # Para quaisquer operações além destas, requeira um usuário autenticado
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    Require valid-user
  </LimitExcept>
</Location>
```

Estes são apenas uns poucos exemplos simples. Para informação mais aprofundada sobre controle de acesso e a diretiva `Require`, dê uma olhada na seção `Security` da coleção de tutoriais da documentação do Apache em http://httpd.apache.org/docs-2.0/misc/tutorials.html.

Controle de Acesso por Diretório

É possível configurar permissões mais granularizadas usando um segundo módulo do Apache httpd, **mod_authz_svn**. Este módulo captura várias URLs opacas passando do cliente para o servidor, pede ao **mod_dav_svn** para decodificá-las, e então possivelmente restringe requisições baseadas em políticas de acesso definidas em um arquivo de configuração.

Se você compilou o Subversion a partir do código-fonte, o **mod_authz_svn** é construído automaticamente e instalado juntamente com o **mod_dav_svn**. Muitas distribuições binárias também o instalam automaticamente. Para verificar se está instalado corretamente, assegure-se de que ele venha logo depois da diretiva `LoadModule` do **mod_dav_svn** no `httpd.conf`:

```
LoadModule dav_module         modules/mod_dav.so
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

```
LoadModule authz_svn_module   modules/mod_authz_svn.so
```

Para ativar este módulo, você precisa configurar seu bloco `Location` para usar a diretiva `AuthzSVNAccessFile`, a qual especifica um arquivo contendo políticas de permissões para caminhos dentro de seus repositórios. (Logo, logo, vamos discutir o formato deste arquivo.)

O Apache é flexível, então você tem a opção de configurar seu bloco em um destes três padrões gerais. Para começar, escolha um destes três padrões de configuração. (Os exemplos abaixo são muito simples; consulte a documentação do próprio Apache para ter muito mais detalhes sobre as opções de autenticação e autorização do Apache.)

O bloco mais simples é permitir acesso abertamente a todo mundo. Neste cenário, o Apache nunca envia desafios de autenticação, então todos os usuários são tratados como "anonymous".

Exemplo 6.1. Um exemplo de configuração para acesso anônimo.

```
<Location /repos>
  DAV svn
  SVNParentPath /usr/local/svn

  # nossa política de controle de acesso
  AuthzSVNAccessFile /caminho/do/arquivo/access
</Location>
```

No outro extremo da escala de paranóia, você pode configurar seu bloco para requisitar autenticação de todo mundo. Todos os clientes devem prover suas credenciais para se identificarem. Seu bloco requer autenticação incondicional através da diretiva `Require valid-user`, e define meios para autenticação.

Exemplo 6.2. Um exemplo de configuração para acesso autenticado.

```
<Location /repos>
  DAV svn
  SVNParentPath /usr/local/svn

  # nossa política de controle de acesso
  AuthzSVNAccessFile /caminho/do/arquivo/access

  # apenas usuários autenticados podem acessar o repositório
  Require valid-user

  # como autenticar um usuário
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /caminho/do/arquivo/users
</Location>
```

Um terceiro padrão bem popular é permitir uma combinação de acesso autenticado e anônimo. Por exemplo, muitos administradores querem permitir usuários anônimos a ler certos diretórios do repositório, mas querem que apenas usuários autenticados leiam (ou escrevam) em áreas mais sensíveis. Nesta configuração, todos os usuários começam acessando o repositório anonimamente. Se sua política de controle de acesso solicitar um nome de usuário real em algum ponto, o Apache vai solicitar autenticação para o cliente. Para fazer isto, você usa ambas as diretivas `Satisfy Any` e `Require valid-user` em conjunto.

Exemplo 6.3. Um exemplo de configuração para acesso misto autenticado/anônimo.

```
<Location /repos>
  DAV svn
  SVNParentPath /usr/local/svn

  # nossa política de controle de acesso
  AuthzSVNAccessFile /caminho/do/arquivo/access

  # tente o acesso anônimo primeiro, ajuste para autenticação
  # real se necessário
  Satisfy Any
  Require valid-user

  # como autenticar um usuário
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /caminho/do/arquivo/users
</Location>
```

Tendo se decidido por um destes três modelos básicos de `httpd.conf`, você então precisa criar seu arquivo contendo regras de acesso para determinados caminhos dentro do repositório. Isto é descrito em “Autorização Baseada em Caminhos”.

Desabilitando Verificação baseada em Caminhos

O módulo **mod_dav_svn** realiza bastante trabalho para se assegurar de que os dados que você marcou como “unreadable” não sejam corrompidos acidentalmente. Isto significa que ele precisa monitorar de perto todos os caminhos e conteúdos de arquivos retornados por comandos como **svn checkout** ou **svn update**. Se estes comandos encontram um caminho que não seja legível de acordo com alguma política de autorização, então, tipicamente, o caminho como um todo é omitido. No caso de histórico ou acompanhamento de renomeações—p.ex. ao se executar um comando como **svn cat -r OLD foo.c** em um arquivo que foi renomeado há bastante tempo—o acompanhamento da renomeação irá simplesmente parar se um dos nomes anteriores do objeto for determinado como sendo de leitura restrita.

Toda esta verificação de caminhos algumas vezes pode ser um pouco custosa, especialmente no caso do **svn log**. Ao obter uma lista de revisões, o servidor olha para cada caminho alterado em cada revisão e verifica sua legibilidade. Se um caminho ilegível é descoberto, então ele é omitido da lista de caminhos alterados da revisão (normalmente vista com a opção `--verbose`), e a mensagem de log completa é suprimida. Desnecessário dizer que isto pode consumir bastante tempo em revisões que afetam um grande número de arquivos. Este é o custo da segurança: mesmo se você ainda nunca tiver configurado um módulo como **mod_authz_svn**, o módulo **mod_dav_svn** ainda fica solicitando para que o Apache **httpd** execute verificações de autorização em cada caminho. O módulo **mod_dav_svn** não faz idéia de que módulos de autorização estão instalados, então tudo o que ele pode fazer é solicitar que o Apache invoque todos os que podem estar presentes.

Por outro lado, também há um recurso de válvula de escape, o qual permite a você troque características de segurança por velocidade. Se você não estiver impondo nenhum tipo de autorização por diretório (i.e. não está usando o módulo **mod_authz_svn** ou similar), então você pode desabilitar toda esta checagem de caminhos. Em seu arquivo `httpd.conf`, use a diretiva `SVNPathAuthz`:

Exemplo 6.4. Desabilitando verificações de caminho como um todo

```
<Location /repos>
  DAV svn
  SVNParentPath /usr/local/svn

  SVNPathAuthz off
</Location>
```

A diretiva SVNPathAuthz é definida como “on” por padrão. Quando definida para “off”, toda a checagem de autorização baseada em caminhos é desabilitada; o **mod_dav_svn** pára de solicitar checagem de autorização em cada caminho que ele descobre.

Facilidades Extras

Cobrimos a maior parte das opções de autenticação e autorização para o Apache e o mod_dav_svn. Mas há alguns outros poucos bons recursos que o Apache provê.

Navegação de Repositório

Um dos mais úteis benefícios de uma configuração do Apache/WebDAV para seu repositório Subversion é que as revisões mais recentes de seus arquivos e diretórios sob controle de versão ficam imediatamente disponíveis para visualização por meio de um navegador web comum. Como o Subversion usa URLs para identificar recursos sob controle de versão, estas URLs usadas para acesso ao repositório baseado em HTTP podem ser digitadas diretamente num navegador Web. Seu navegador vai emitir uma requisição HTTP GET para aquela URL, e se aquela URL representar um diretório ou arquivo sobre controle de versão, o mod_dav_svn irá responder com uma listagem de diretório ou com o conteúdo do arquivo.

Já que as URLs não contém nenhuma informação sobre quais versões do recurso você quer ver, o mod_dav_svn sempre irá responder com a versão mais recente. Esta funcionalidade tem um maravilhoso efeito colateral que é a possibilidade de informar URLs do Subversion a seus parceiros como referências aos documentos, e estas URLs sempre vão apontar para a versão mais recente do documento. É claro, você ainda pode usar URLs como hiperlinks em outros web sites, também.

Posso ver revisões antigas?

Com um navegador web comum? Em uma palavra: não. Ao menos, não com o **mod_dav_svn** como sua única ferramenta.

Seu navegador web entende apenas HTTP padrão. Isso significa que ele apenas sabe como obter (GET) URLs públicas, as quais representam as últimas versões dos arquivos e diretórios. De acordo com a especificação WebDAV/DeltaV, cada servidor define uma sintaxe de URL particular para versões mais antigas dos recursos, e esta sintaxe é opaca aos clientes. Para encontrar uma versão mais antiga de um arquivo, um cliente deve seguir um processo específico para “descobrir” a URL adequada; o procedimento envolve um conjunto de requisições PROPFIND do WebDAV e a compreensão de conceitos do DeltaV. Isto é algo que seu navegador web simplesmente não consegue fazer.

Então para responder à pergunta, a maneira óbvia de ver revisões antigas de arquivos e diretórios é pela passagem do argumento `--revision (-r)` para os comando **svn list** e **svn cat**. Para navegar em revisões antigas com seu navegador web, entretanto, você precisar usar software de terceiros. Um bom exemplo disto é o ViewVC (http://viewvc.tigris.org/). Originalmente o ViewVC foi escrito para exibir repositórios CVS pela web, [7] mas as versões mais recentes trabalham com repositórios Subversion, também.

[7] No início, ele chamava-se “ViewCVS”.

Tipo MIME Adequado

Ao navegar em um repositório Subversion, o navegador web obtém um indício sobre como renderizar o conteúdo do arquivo consultando o cabeçalho `Content-Type:` retornado na resposta da requisição HTTP `GET`. O valor deste cabeçalho é o valor de um tipo MIME. Por padrão, o Apache vai indicar aos navegadores web que todos os arquivos do repositório são do tipo MIME “default”, usualmente o tipo `text/plain`. Isto pode ser frustrante, entretanto, se um usuário quiser que os arquivos do repositório sejam renderizados como algo com mais significado—por exemplo, seria ótimo que um arquivo `foo.html` pudesse ser renderizado como HTML na navegação.

Para fazer isto acontecer, você só precisa garantir que seus arquivos tenham o `svn:mime-type` adequadamente configurado. Isto é discutido em mais detalhes em “Tipo de Conteúdo do Arquivo”, a você ainda pode configurar seu cliente para anexar as propriedades `svn:mime-type` automaticamente aos arquivos que estejam entrando no repositório pela primeira vez; veja “Definição Automática de Propriedades”.

Assim, em nosso exemplo, se alguém definiu a propriedade `svn:mime-type` para `text/html` no arquivo `foo.html`, então o Apache deve avisar adequadamente para que seu navegador web renderize o arquivo como HTML. Alguém também poderia anexar propriedades `image/*` de tipos mime para imagens, e fazendo isso, no final das contas obter um site web completo podendo ser visualizado diretamente do repositório! Geralmente não há problema em se fazer isto, desde que o website não contenha nenhum conteúdo gerado dinamicamente.

Personalizando a Aparência

Você normalmente vai fazer mais uso das URLs para arquivos versionados—afinal, é onde o conteúdo interessante tende a estar. Mas pode haver certas situações você pode precisar navegar na listagem de diretórios, no que você rapidamente irá notar que o HTML gerado para exibir estas listagens é muito básico, e certamente não pretende ser esteticamente agradável (ou mesmo interessante). Para possibilitar a personalização destas exibições de diretório, o Subversion provê um recurso de índice em XML. Uma única diretiva `SVNIndexXSLT` no bloco `Location` do seu `httpd.conf` vai orientar o mod_dav_svn a gerar saída XML ao exibir uma listagem de diretório, e referenciar uma folha de estilos XSLT à sua escolha:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  SVNIndexXSLT "/svnindex.xsl"
  …
</Location>
```

Usando a diretiva `SVNIndexXSLT` e uma folha de estilos criativa, você pode fazer com que suas listagens de diretórios sigam os esquemas de cores e imagens usados em outras partes de seu website. Ou, se você preferir, você pode usar folhas de estilo de exemplo que já vêm no diretório `tools/xslt` dos fontes do Subversion. Tenha em mente que o caminho informado para o diretório em `SVNIndexXSLT` atualmente é um caminho de URL—os navegadores precisam conseguir ler suas folhas de estilo para que possam fazer uso delas!

Listando Repositórios

Se você está servindo um conjunto de repositórios a partir de uma única URL por meio da diretiva `SVNParentPath`, então também é possível fazer o Apache exibir todos os repositórios disponíveis para o navegador web. Apenas ative a diretiva `SVNListParentPath`:

```
<Location /svn>
  DAV svn
```

```
  SVNParentPath /usr/local/svn
  SVNListParentPath on
  …
</Location>
```

Se um usuário agora apontar seu navegador web para a URL `http://host.example.com/svn/`, ele irá ver uma lista de todos os repositórios Subversion situados em `/usr/local/svn`. É claro que isto pode representar um problema de segurança, por isso este recurso é desabilitado por padrão.

Logs do Apache

Pelo fato de o Apache ser um servidor HTTP por vocação, ele possui recursos para log fantasticamente flexíveis. Está além do escopo deste livro discutir todas as formas de log que podem ser configuradas, mas ressaltamos que mesmo o arquivo `httpd.conf` mais genérico vai fazer com que o Apache produza dois arquivos de log: `error_log` e `access_log`. Estes logs podem ficar em diferentes lugares, mas comumente são criados na área de logs de sua instalação do Apache. (No Unix, eles frequentemente ficam em `/usr/local/apache2/logs/`.)

O `error_log` descreve quaisquer erros internos que ocorram com o Apache durante sua execução. Já o arquivo `access_log` faz um registro de cada requisição recebida pelo Apache. Com isto fica fácil de ver, por exemplo, quais endereços IP dos clientes Subversion que estão se conectando, com que frequência os clientes acessam o servidor, quais usuários estão devidamente autenticados, e quantas requisições tiveram sucesso ou falharam.

Infelizmente, como o HTTP é um protocolo sem informação de estado, mesmo a mais simples operação de um cliente Subversion gera múltiplas requisições de rede. É muito difícil olhar o arquivo `access_log` e deduzir o que o cliente estava fazendo—muitas operações se parecem uma série de obscuras requisições `PROPPATCH`, `GET`, `PUT`, e `REPORT`. Para piorar as coisas, muitas operações dos clientes enviam conjuntos de requisições quase idênticas, então é ainda mais difícil diferenciá-las.

O `mod_dav_svn`, entretanto, pode ajudar. Ativando um recurso de "log operacional", você pode pode instruir o `mod_dav_svn` a criar arquivos de log separados os tipos de operações em alto nível que os clientes estão executando.

Para fazer isso, você precisa usar a diretiva `CustomLog` do Apache (a qual é explicada em mais detalhes na própria documentação do Apache). Cuide para invocar esta diretiva *fora* do seu bloco `Location` do Subversion:

```
<Location /svn>
  DAV svn
  …
</Location>

CustomLog logs/svn_logfile "%t %u %{SVN-ACTION}e" env=SVN-ACTION
```

Neste exemplo, solicitamos ao Apache que crie um arquivo de log especial `svn_logfile` no diretório de `logs` padrão do Apache. As variáveis `%t` e `%u` são substituídas, respectivamente, pela hora e pelo nome do usuário da requisição. A parte realmente importante aqui são as duas instâncias de `SVN-ACTION`. Quando o Apache vê esta variável, ele substitui o valor da variável de ambiente `SVN-ACTION`, que é atribuída automaticamente pelo `mod_dav_svn` sempre que ele detecta uma ação do cliente em alto nível.

Assim, ao invés de ter de interpretar um `access_log` tradicional como este:

```
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc/!svn/vcc/default HTTP/1.1" 207 398
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc/!svn/bln/59 HTTP/1.1" 207 449
```

```
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc HTTP/1.1" 207 647
[26/Jan/2007:22:25:29 -0600] "REPORT /svn/calc/!svn/vcc/default HTTP/1.1" 200 607
[26/Jan/2007:22:25:31 -0600] "OPTIONS /svn/calc HTTP/1.1" 200 188
[26/Jan/2007:22:25:31 -0600] "MKACTIVITY /svn/calc/!svn/act/e6035ef7-5df0-4ac0-b811-4be
…
```

… você pode examinar um `svn_logfile` bem mais inteligível como este:

```
[26/Jan/2007:22:24:20 -0600] - list-dir '/'
[26/Jan/2007:22:24:27 -0600] - update '/'
[26/Jan/2007:22:25:29 -0600] - remote-status '/'
[26/Jan/2007:22:25:31 -0600] sally commit r60
```

Outros Recursos

Vários recursos já providos pelo Apache como um servidor Web robusto que é também podem ser aproveitadas para aprimorar as funcionalidades ou a segurança do Subversion. O Subversion se comunica com o Apache usando o Neon, que é uma biblioteca HTTP/WebDAV genérica com suporte a alguns mecanismos como o SSL (*Secure Socket Layer*, discutida anteriormente). Se seu cliente Subversion tiver sido compilado com suporte a SSL, então ele pode acessar seu servidor Apache usando `https://`.

Igualmente útil são outros recursos da relação entre o Apache e o Subversion, como a capacidade de se especificar uma porta específica (ao invés da porta 80, padrão HTTP) ou um nome de domínio virtual a partir do qual um repositório Subversion pode ser acessado, ou a capacidade de se acessar o repositório através de um proxy HTTP. Tudo isto é suportado pelo Neon, então, por tabela, o Subversion tem suporte a tudo isto também.

Finalmente, como o **mod_dav_svn** está falando um subconjunto do protocolo WebDAV/DeltaV, é possível acessar o repositório por meio de clientes DAV de terceiros. A maioria dos sistemas operacionais modernos (Win32, OS X, e Linux) têm a capacidade de montar um servidor DAV como um compartilhamento de rede. Este é um assunto complicado; para mais detalhes, leia Apêndice C, *WebDAV e Autoversionamento*.

Autorização Baseada em Caminhos

Tanto o Apache como o **svnserve** são capazes de garantir (ou negar) permissões aos usuários. Tipicamente isto é feito considerando todo o repositório: um usuário pode ler o repositório (ou não), e pode escrever no repositório (ou não). No entanto, também é possível definir regras de acesso mais pormenorizadas. Um conjunto de usuários podem ter permissão para escrever em um certo diretório do repositório, mas não em outros; outro diretório pode não ser legível por todos, exceto alguns usuários em particular.

Ambos os servidores usam um formato de arquivo comum para descrever estas regras de acesso baseadas em caminhos. No caso do Apache, precisa-se carregar o módulo **mod_authz_svn** e então adicionar-se a diretiva `AuthzSVNAccessFile` (dentro do arquivo `httpd.conf`) para apontar para seu arquivo de regras. (Para uma descrição completa, veja “Controle de Acesso por Diretório”.) Se você está usando o **svnserve**, então você precisa fazer a variável `authz-db` (dentro do `svnserve.conf`) apontar para seu arquivo de regras.

Você realmente precisa de controle de acesso baseado em caminhos?

Muitos administradores que configuram o Subversion pela primeira vez tendem a usar controle de acesso baseado em caminhos mesmo sem pensar muito sobre ele. O administrador comumente sabe quais equipes de pessoas estão trabalhando em quais projetos, então é fácil considerar isso e permitir que certas equipes acessem determinados diretórios e não outros. Parece uma coisa natural, e isso até tranquiliza os desejos dos administradores de manter um controle rígido do repositório.

Perceba, porém, que sempre há custos invisíveis (e visíveis!) associados a este recurso. Quanto aos custos visíveis, tem-se que o servidor precisa de muito mas esforço para garantir que cada usuário tenha o acesso correto de leitura ou escrita em cada caminho específico; em certas circunstâncias, há uma sensível perda de desempenho. Quanto aos custos invisíveis, considere a cultura que você está criando. Na maior parte do tempo, ainda que certos usuários *não devessem* estar registrando alterações em certas partes do repositório, este contrato social não precisa ser reforçado tecnologicamente. Algumas vezes as equipes podem colaborar umas com as outras espontaneamente; alguém pode querer ajudar a algum outro fazendo alterações em alguma parte na qual não trabalha normalmente. Ao prevenir este tipo de coisa a nível de servidor, você está criando inesperadas barreiras à colaboração. Você também está criando um monte de regras que deverão ser mantidas conforme os projetos são desenvolvidos, novos usuários são adicionados, e por aí vai. É muito trabalho extra para manter.

Lembre-se de que isto é um sistema de controle de versão! Mesmo que alguém acidentalmente faça alguma alteração em algo que não deveria, é fácil desfazer a alteração. E se um usuário registrar uma modificação intencionalmente no lugar errado, isto é um problema social de qualquer maneira, e este é um problema que precisará ser tratado fora do Subversion.

Então antes de começar a restringir os direitos de acesso dos usuários, pergunte a si mesmo se há uma razão real e legítima para isto, ou se não é algo que apenas “parece uma boa idéia” para um administrador. Decida ainda se vale a pena sacrificar um pouco da velocidade do servidor, e lembre-se que há muito pouco risco envolvido; é ruim se tornar dependente da tecnologia como uma muleta para problemas sociais[8].

Como um exemplo a considerar, veja que o próprio projeto Subversion sempre teve a noção de quem tem permissão para realizar alterações em que lugares, mas isto já é o que acaba ocorrendo na prática. Este é um bom modelo de confiança da comunidade, especialmente para projetos *open-source*. De fato, algumas vezes *há* razões verdadeiramente legítimas para se ter controle de acesso baseado em caminhos; em empresas, por exemplo, certos tipos de dados realmente podem ser sensíveis, aos quais os acessos precisam ser verdadeiramente restritos a pequenos grupos de pessoas.

Uma vez que o servidor saiba onde encontrar seu arquivo de regras, é hora de definí-las.

A sintaxe do arquivo é aquela mesma sintaxe familiar usada no **svnserve.conf** e nos arquivos de configuração em tempo de execução. Linhas que comecem com cerquilha (#) são ignoradas. Em sua forma mais simples, cada seção nomeia um repositório e um caminho dentro dele, e os nomes de usuários autenticados são os nomes das opções dentro de cada seção. O valor de cada opção descreve o nível de acesso dos usuários naquele caminho do repositório: seja `r` (somente leitura) ou `rw` (leitura/escrita). Se o usuário não for mencionado de forma nenhuma, nenhum acesso será permitido.

Para ser mais específico: o valor dos nomes das seções ou são da forma `[repos-name:path]` ou da forma `[path]`. Se você está usando a diretiva `SVNParentPath`, então é importante especificar os nomes dos repositórios em suas seções. Se você omití-los, então uma seção como `[/algum/dir]` irá corresponder ao caminho `/algum/dir` em *cada* repositório. Se você está usando a diretiva `SVNPath`, porém, então não há problema em definir apenas os caminhos em suas seções—afinal de contas, há apenas um repositório.

[8]Um tema recorrente neste livro!

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r
```

Neste primeiro, o usuário `harry` tem completo acesso de leitura e escrita ao diretório `/branches/calc/bug-142` no repositório `calc`, mas o usuário `sally` tem acesso somente leitura. Quaisquer outros usuários têm seu acesso a este repositório bloqueado.

É claro que as permissões são herdadas de um diretório para um filho. Isto quer dizer que podemos especificar um subdiretório com uma política de acesso diferente para Sally:

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r

# dá à sally acesso de escrita apenas no subdiretório 'testing'
[calc:/branches/calc/bug-142/testing]
sally = rw
```

Agora Sally pode escrever no subdiretório `testing` do ramo, mas ainda continua tendo acesso somente leitura a outras partes. Harry, no entanto, continua a ter acesso completo de leitura/escrita ao ramo inteiro.

Também é possível negar permissão a alguns usuários através das regras de herança, removendo o valor da variável do nome do usuário:

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r

[calc:/branches/calc/bug-142/secret]
harry =
```

Neste exemplo, Harry tem acesso completo leitura/escrita à toda a árvore `bug-142`, mas não tem absolutamente nenhum acesso em todo o subdiretório `secret` dentro dela.

O que você deve lembrar é que a correspondência sempre é feita com os caminhos mais específicos primeiro. O servidor tenta achar uma ocorrência com o próprio caminho, então depois com o caminho do diretório pai, e depois com o pai deste, e assim por diante. O efeito em rede é que mencionando um caminho específico no arquivo de acesso sempre irá sobrescrever qualquer permissão herdada dos diretórios pais.

Por padrão, ninguém tem acesso ao repositório como um todo. Isto significa que se você está iniciando com um arquivo vazio, você provavelmente quer pelo menos dar permissão de leitura a todos os usuários na raiz do repositório. Você pode fazer isso usando a variável asterisco (`*`), o que quer dizer “todos os usuários”:

```
[/]
* = r
```

Esta é uma configuração comum; note que não aparece o nome de nenhum repositório no nome da seção. Isto torna todos os repositórios legíveis para todos os usuários. Uma vez que todos os usuários tem acesso de leitura aos repositórios, você pode dar permissões `rw` explícitas a certos usuários em subdiretórios dentro de repositórios específicos.

A variável asterisco (`*`) merece também um destaque especial aqui: é o *único* padrão que corresponde com o usuário anônimo. Se você configurou seu bloco servidor para permitir um misto de acesso

anônimo e autenticado, todos os usuários iniciam acessando anonimamente. O servidor procura por um valor * definido para o caminho sendo acessado; se encontrar, então requisita autenticação efetiva do cliente.

O arquivo de acesso também lhe possibilita definir grupos inteiros de usuários, tal como o arquivo /etc/group do Unix:

```
[groups]
calc-developers = harry, sally, joe
paint-developers = frank, sally, jane
everyone = harry, sally, joe, frank, sally, jane
```

Controle de acesso pode ser definido para grupos da mesma forma como para usuários. Sendo que os grupos se distinguem por tem um sinal de “arroba” (@) na frente:

```
[calc:/projects/calc]
@calc-developers = rw

[paint:/projects/paint]
@paint-developers = rw
jane = r
```

Grupos também podem ser definidos de forma a conter outros grupos:

```
[groups]
calc-developers = harry, sally, joe
paint-developers = frank, sally, jane
everyone = @calc-developers, @paint-developers
```

Legibilidade Parcial e Checkouts

Se você está usando o Apache como seu servidor Subversion e deixou determinados subdiretórios de seu repositório ilegíveis para certos usuários, então você precisa ter cuidado com um possível comportamento não-otimizado do comando **svn checkout**.

Quando o cliente realiza um checkout ou update sobre HTTP, ele faz uma única requisição ao servidor, e recebe uma única resposta (quase sempre bem grande). Quando o servidor recebe a requisição, que é a *única* oportunidade que o Apache tem de solicitar autenticação do usuário. Isto tem alguns efeitos colaterais. Por exemplo, se um certo determinado subdiretório do repositório é legível apenas pelo usuário Sally, e o usuário Harry dá um checkout num diretório pai, seu cliente vai atender ao desafio de autenticação inicial como Harry. Como o servidor gera uma resposta grande, não há uma forma de reenviar um desafio de autenticação quando encontrar um subdiretório especial; pulando assim este subdiretório como um todo, em vez de solicitar que o usuário se re-autentique como Sally no momento certo. De maneira similar, se a raiz do repositório é legível anonimamente por todos, então todo checkout será feito sem autenticação—novamente, pulando o diretório legível em vez de solicitar autenticação para ter acesso a essas partes do repositório.

Dando Suporte a Múltiplos Métodos de Acesso ao Repositório

Você viu como um repositório pode ser acessado de diferentes maneiras. Mas também é possível—ou seguro—que seu repositório seja acessado por meio de diferentes métodos ao mesmo tempo? A resposta é sim, desde que você seja um pouco previdente.

A qualquer momento, estes processos podem demandar acesso de leitura e escrita ao seu repositório:

- usuários regulares do sistema, usando um cliente Subversion (como si próprios) para acessar o repositório diretamente por meio de URLs `file://`;
- usuários regulares do sistema se conectando a processos **svnserve** particulares (executando como si próprios) que acessam o repositório;
- um processo **svnserve**—seja um daemon ou disparado pelo **inetd**—executando como um determinado usuário em particular;
- um processo Apache **httpd**, executando como um usuário em particular.

O problema mais comum que os administradores enfrentam diz respeito a propriedade e a permissões do repositório. Cada um dos processos (ou usuários) da lista anterior tem direito de ler e escrever nos arquivos Berkeley DB da base de dados? Assumindo que você esteja num sistema operacional Unix-like, uma abordagem simples poderia ser colocar cada usuário do repositório em potencial em um novo grupo `svn`, e fazer com que o repositório inteiro pertença a este grupo. Mas isso ainda não é o suficiente, porque um processo pode escrever nos arquivos da base de dados usando um valor de umask problemático—que impossibilite o acesso de outros usuários.

Então além de definir um grupo comum para os usuários do repositório, o próximo passo é forçar que cada processo que acesse o repositório use um valor adequado de umask. Para usuários que acessam o repositório diretamente, você pode transformar o programa **svn** em um script que primeiro defina **umask 002** e então execute o programa cliente **svn** real. Você pode escrever um script semelhante para o programa **svnserve**, e adicionar um comando **umask 002** ao próprio script de script de inicialização do Apache, `apachectl`. Por exemplo:

```
$ cat /usr/bin/svn

#!/bin/sh

umask 002
/usr/bin/svn-real "$@"
```

Outro problema comum é frequentemente encontrado em sistemas Unix-like. Conforme um repositório é usado, o Berkeley DB ocasionalmente cria novos arquivos de log para registrar suas ações. Mesmo num repositório que pertença inteiramente ao grupo **svn**, estes novos arquivos criados não pertencerão necessariamente a este grupo, o que então cria mais problemas de permissão para seus usuários. Uma boa forma de contornar isto é definir o bit SUID dos diretórios db do repositório. Isto faz com que todos os arquivos recém-criados pertençam ao mesmo grupo de seu diretório pai.

Uma vez realizados estes passos, seu repositório deve ser acessível para todos os processos necessários. Pode parecer um pouco confuso e complicado, mas os problemas de ter múltiplos usuários compartilhando acesso de escrita a arquivos comuns são problemas clássicos e que muitas vezes não são resolvidos de maneira muito elegante.

Felizmente, muitos administradores de repositórios nunca irão *precisar* realizar tão complexa configuração. Os usuários que querem acessar repositórios que estejam na mesma máquina não estão limitados a usar URLs `file://` para acesso—eles normalmente podem contactar um servidor Apache HTTP ou **svnserve** usando `localhost` como nome do servidor em suas URLs `http://` ou `svn://`. E manter múltiplos processos servidores para seus repositórios Subversion é estar propenso a mais dor de cabeça que o necessário. Nós recomendamos que você escolha o servidor que melhor atenda às suas necessidades e siga firme com ele!

Um checklist para o servidor svn+ssh://

Pode ser um pouco complicado fazer com que uma porção de usuários com contas SSH existentes compartilhem um repositório sem problemas de permissão. Se você está confuso sobre todas as coisas que você (como um administrador) precisa fazer em um sistema Unix-like, aqui está um breve checklist que resume algumas das coisas discutidas nesta seção:

- Todos os seus usuários SSH precisam ser capazes de ler e escrever no repositório, então: ponha todos os usuários SSH em um mesmo grupo.
- Faça com que o repositório inteiro pertença a esse grupo.
- Defina as permissões de grupo para leitura/escrita.
- Seus usuários precisar usar um valor de umask adequado ao acessar o repositório, então: confirme que o **svnserve** (`/usr/bin/svnserve`), ou onde quer que ele esteja no `$PATH`) seja atualmente um script que encapsule **umask 002** e execute o binário **svnserve** real.
- Tome medidas similares ao usar **svnlook** e **svnadmin**. Ou execute-os com um umask adequado, ou encapsule-os conforme descrito acima.

Capítulo 7. Customizando sua Experiência com Subversion

Controle de versão pode ser um tema complexo, assim como arte ou ciência, e oferecer variadas maneiras fazer as coisas. Através desse livro você leu sobre vários subcomandos do cliente de linha de comando e as opções para modificar seu comportamento. Nesse capítulo, vamos dar uma olhada e mais maneiras de customizar o jeito que o Subversion trabalha para você—definindo as configurações em tempo de execução, usando ajuda de aplicações externas, a interação do Subversion com as configurações locais do sistema operacional, e assim por diante.

Área de Configuração do Tempo de Execução

O Subversion oferece muitos comportamentos opcionais que podem ser controlados pelo usuário. Muitas dessas opções são do tipo que um usuário desejaria aplicar a todas as operações do Subversion. Então, em vez de obrigar os usuários lembrar de argumentos de linha de comando para especificar essas opções, e usá-las para toda operação que ele realizar, o Subversion usa arquivos de configuração, segregadas em uma área de configuração do Subversion.

A *área de configuração* do Subversion é uma hierarquia de opções com nomes e seus valores, em dois níveis. Normalmente, isto resume-se a um diretório especial que contém *arquivos de configuração* (o primeiro nível), os quais são apenas arquivos de texto no formato padrão INI (com “seções” provendo o segundo nível). Estes arquivos podem ser facilmente editados usando seu editor de texto favorito (como Emacs ou vi), e contém diretivas que são lidas pelo cliente para determinar quais dos vários comportamentos opcionais o usuário prefere.

Estrutura da Área de Configuração

A primeira vez que o cliente de linha de comando **svn** é executado, ele cria uma área de configuração por usuário. Em sistemas Unix, esta área aparece como um diretório chamado `.subversion` no diretório pessoal do usuário. Em sistemas Win32, o Subversion cria uma pasta chamada `Subversion`, geralmente dentro da área `Application Data` do diretório do perfil do usuário (que, por padrão, é normalmente um diretório oculto). No entanto, nesta plataforma o local exato difere de sistema para sistema e é definido pelo Registro do Windows. [1] Nós referiremos à área de configuração por usuário usando seu nome em Unix, `.subversion`.

Além da área de configuração por usuário, o Subversion também reconhece a existência de uma área de configuração a nível de sistema. Isto oferece aos administradores de sistema a habilidade de estabelecer padrões para todos usuários em uma determinada máquina. Note que a área de configuração a nível de sistema não define uma política obrigatória—as definições na área de configuração do usuário substitui as de nível de sistema, e os argumentos de linha de comando passados ao programa **svn** possuem a palavra final no comportamento realizado. Em plataformas Unix, espera-se que a área de configuração do sistema esteja no diretório `/etc/subversion`; em máquinas Windows, ela aparece em um diretório `Subversion` dentro do local `Application Data` do sistema (novamente, como especificado pelo Registro do Windows). Diferentemente do caso por usuário, o programa **svn** não tenta criar a área de configuração a nível de sistema.

A área de configuração por usuário atualmente contém três arquivos—dois arquivos de configuração (`config` e `servers`), e um arquivo `README.txt` que descreve o formato INI. Quando são criados, os arquivos contém valores padrão para cada uma das opções suportadas pelo Subversion, a maioria

[1] A variável de ambiente APPDATA aponta para a área `Application Data`, assim você sempre pode referir a esta pasta como `%APPDATA%\Subversion`.

comentadas e agrupadas com descrições textuais sobre como os valores para as chaves afetam o comportamento do Subversion. Para mudar um certo comportamento, você apenas precisa carregar o arquivo de configuração apropriado em um editor de texto, e modificar o valor da opção desejada. Se, a qualquer momento, você desejar ter as definições da configuração padrão restauradas, você pode simplesmente remover (ou renomear) seu diretório de configuração e então executar algum comando **svn** inofensivo, como o **svn --version**. Um novo diretório de configuração com o conteúdo padrão será criado.

A área de configuração por usuário também contém uma cache dos dados de autenticação. O diretório `auth` possui um conjunto de subdiretórios que contém pedaços das informações armazenadas e usadas pelos vários métodos de autenticação suportados pelo Subversion. Este diretório é criado de tal forma que somente o próprio usuário possui permissão para ler seu conteúdo.

Configuração e o Registro do Windows

Além da usual área de configuração baseada em arquivos INI, clientes Subversion que executam sob plataformas Windows podem também podem usar o registro do Windows para armazenar dados de configuração. Os nomes das opções e seus valores são os mesmos que os dos arquivos INI. A hierarquia “arquivo/seção” também é preservada, ainda que abordada de maneira ligeiramente diferente—neste esquema, arquivos são apenas níveis na árvore de chaves do registro.

O Subversion procura por valores de configuração globais em nível de sistema sob a chave `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion`. Por exemplo, a opção `global-ignores`, a qual está está na seção `miscellany` do arquivo `config`, poderia ser encontrada em `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Config\Miscellany\global-ignores`. Valores de configuração por usuário devem estar armazenados abaixo de `HKEY_CURRENT_USER\Software\Tigris.org\Subversion`.

Opções de configuração baseadas em registro são analisadas *antes* do que suas contrapartes em arquivo, assim elas são sobrescritas pelos valores encontrados nos arquivos de configuração. Em outras palavras, o Subversion procura por informação de configuração nos seguintes locais em um sistema Windows; a relação abaixo organiza os locais em ordem dos de maior para os de menor precedência:

1. Opções de linha de comando

2. Arquivos INI por usuário

3. Valores de registro por usuário

4. Arquivos INI globais em nível de sistema

5. Valores de registro em nível de sistema

Adicionalmente, o registro do Windows não suporta a noção de algo sendo “comentado”. No entanto, o Subversion irá ignorar quaisquer opções cujos nome da chave comece com um caractere de cerquilha (#). Na prática, efetivamente isto permite que você comente uma opção do Subversion sem remover a chave inteira do registro, o que obviamente simplifica o processo de restaurar tal opção.

O cliente de linha de comando, **svn**, tenta escrever no registro do Windows, e então não tentará escrever uma área de configuração padrão. Você pode criar as chaves que você precisa usando o programa **REGEDIT**. Alternativamente, você pode criar um arquivo `.reg`, e então dar um duplo clique nele a partir do Windows Explorer, o que irá fazer com que os dados sejam mesclados ao seu registro.

Exemplo 7.1. Arquivo (.reg) com Entradas de Registro de Exemplo.

```
REGEDIT4

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\groups]

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\global]
"#http-proxy-host"=""
"#http-proxy-port"=""
"#http-proxy-username"=""
"#http-proxy-password"=""
"#http-proxy-exceptions"=""
"#http-timeout"="0"
"#http-compression"="yes"
"#neon-debug-mask"=""
"#ssl-authority-files"=""
"#ssl-trust-default-ca"=""
"#ssl-client-cert-file"=""
"#ssl-client-cert-password"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auth]
"#store-passwords"="yes"
"#store-auth-creds"="yes"

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\helpers]
"#editor-cmd"="notepad"
"#diff-cmd"=""
"#diff3-cmd"=""
"#diff3-has-program-arg"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\tunnels]

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\miscellany]
"#global-ignores"="*.o *.lo *.la #*# .*.rej *.rej .*~ *~ .#* .DS_Store"
"#log-encoding"=""
"#use-commit-times"=""
"#no-unlock"=""
"#enable-auto-props"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auto-props]
```

O exemplo anterior mostra o conteúdo de um arquivo `.reg` que contém algumas das opções de configuração mais comuns e seus valores padrão. Note a presença tanto de configurações em nível de sistema (para opções relacionadas a proxies de rede) e configurações específicas por usuário (programa editor de texto e armazenamento de senhas, entre outras). Também note que todas as opções estão efetivamente comentadas. Você precisa apenas remover o caractere de cerquilha (#) do começo dos nomes das opções, e definir os valores como você quiser.

Opções de Configuração

Nesta seção, vamos discutir as opções de configuração específicas de tempo de execução que são atualmente suportadas pelo Subversion.

Servidores

O arquivo `servers` contém opções de configuração relacionadas à camada de rede. Há dois nomes de seção neste arquivo—`groups` e `global`. A seção `groups` é essencialmente uma tabela de referência cruzada. As chaves nesta seção são os nomes de outras seções no arquivo; seus valores são *globs*—indicadores textuais que possivelmente podem conter caracteres coringa—que são comparados com os nomes de hosts das máquinas às quais as requisições do Subversion são enviadas.

```
[groups]
beanie-babies = *.red-bean.com
collabnet = svn.collab.net

[beanie-babies]
…

[collabnet]
…
```

Quando o Subversion é usado em rede, ele tenta casar o nome do servidor ao qual está tentando alcançar com o nome do grupo na seção `groups`. Se encontrar uma correspondência, o Subversion então procura por uma seção no arquivo `servers` com o mesmo nome do grupo em questão. A partir dessa seção, ele lê as configurações de rede atuais.

A seção `global` contém as configurações que são feitas para todos os servidores para os quais não haja correspondência na seção `groups`. As opções disponíveis nesta seção são exatamente as mesmas que aquelas válidas para outras seções de servidores no arquivo (exceto, é claro, a seção especial `groups`), e são como o que se segue:

`http-proxy-exceptions`
: Isto especifica uma lista de padrões de nomes de hosts separados por vírgulas e que devem ser acessados diretamente, sem que seja por meio de uma máquina de proxy. A sintaxe desses padrões é a mesma que utilizada no shell do Unix para nomes de arquivos. Um nome de host de um repositório que corresponda a qualquer desses padrões não será acessado através de um proxy.

`http-proxy-host`
: Isto especifica o nome do host do computador proxy através do qual suas requisições HTTP para o Subversion devem passar. Seu padrão é um valor vazio, o que significa que o Subversion não tentará rotear requisições HTTP através de um computador proxy, ao invés disso tentando acessar a máquina de destino diretamente.

`http-proxy-port`
: Isto especifica o número de porta a ser usada no host proxy. Seu valor padrão também é vazio.

`http-proxy-username`
: Isto especifica o nome de usuário a ser informado para a máquina proxy. Seu valor padrão é vazio.

`http-proxy-password`
: Isto especifica a senha do usuário a ser informada para a máquina proxy. Seu valor padrão é vazio.

`http-timeout`
: Isto especifica o total de tempo, em segundos, a aguardar por uma resposta do servidor. Se você tiver problemas com uma conexão de rede lenta e que façam com que as operações do Subversion terminem por expiração do limite de tempo, você deveria aumentar o valor desta opção. O valor padrão é `0`, o que instrui a biblioteca HTTP da camada inferior, a Neon, para usar sua própria configuração de limite de tempo.

`http-compression`
: Isto especifica se o Subversion deve ou não tentar compactar as requisições de rede feitas a servidores compatíveis com DAV. O valor padrão é `yes` (ainda que a compactação só irá ocorrer

se o suporte a este recurso estiver compilado na camada de rede). Atribua a esta opção o valor `no` para desabilitar compactação, por exemplo ao depurar transmissões de rede.

`neon-debug-mask`
: Isto é uma máscara inteira que a biblioteca HTTP da camada inferior, a Neon, utiliza para escolher que tipo de saída de depuração ela deve gerar. O valor padrão é `0`, que irá silenciar toda a saída de depuração. Para mais informações sobre como o Subversion faz uso da biblioteca Neon, veja Capítulo 8, *Incorporando o Subversion*.

`ssl-authority-files`
: Isto é uma lista de caminhos delimitada por pontos-e-vírgulas, para os arquivos que contenham certificados das autoridades certificadoras (as ACs) que são aceitas pelo cliente Subversion quando acessando repositórios sob HTTPS.

`ssl-trust-default-ca`
: Defina esta variável para `yes` se você quer que o Subversion confie automaticamente no conjunto padrão de ACs que vêm com o OpenSSL.

`ssl-client-cert-file`
: Se um host (ou um conjunto de hosts) necessitar de um certificado SSL de cliente, você provavelmente será solicitado a informar um caminho para seu certificado. Ao definir esta variável para o mesmo caminho, o Subversion será capaz de encontrar seu certificado de cliente automaticamente sem precisa lhe solicitar esta informação. Não há um local padronizado para armazenar seu certificado de cliente no disco; o Subversion conseguirá lê-lo a partir de qualquer caminho que você especificar.

`ssl-client-cert-password`
: Se seu arquivo do certificado de cliente SSL estiver criptografado com uma frase-senha, o Subversion irá lhe pedir que você forneça a frase-senha toda vez que certificado for usado. Se você achar isto um estorvo (e se não se importa em deixar a senha armazenada no arquivo `servers`), então você pode definir esta variável para a frase-senha do certificado. Assim, esta informação não mais lhe será solicitada.

Configuração

O arquivo `config` contém as demais opções em tempo de execução atualmente disponíveis no Subversion, estas não relacionadas à conexão de rede. Há apenas algumas poucas opções em uso até o momento em que este livro estava sendo escrito, mas elas ainda são agrupadas em seções na expectativa para adições futuras.

A seção `auth` contém configurações relacionadas à autenticação e autorização no Subversion no repositório. Ela contém:

`store-passwords`
: Isto orienta o Subversion a armazenar, ou não, em cache as senhas que são informadas pelo usuário em resposta a desafios de autenticação do servidor. O valor padrão é `yes`. Altere seu valor para `no` para desabilitar este recurso de cache de senhas em disco. Você pode sobrescrever esta opção para uma única instância do comando **svn** usando o parâmetro `--no-auth-cache` de linha de comando (para os subcomandos que dêem suporte a ele). Para mais informações, veja “Armazenando Credenciais no Cliente”.

`store-auth-creds`
: Esta configuração é idêntica a `store-passwords`, exceto que ela habilita ou desabilita o cache em disco de *todas* as informações de autenticação: nomes de usuário, senhas, certificados de servidor, e quaisquer outros tipos de credenciais armazenáveis.

A seção `helpers` controla quais aplicações externas o Subversion usa como suporte na execução de suas tarefas. Opções válidas nesta seção são:

`editor-cmd`
: Isto especifica o programa que o Subversion irá usar para solicitar uma mensagem longa ao usuário durante uma operação de submissão, como ao usar **svn commit** sem as opções `--message` (`-m`) ou `--file` (`-F`). Este programa também é usado com o comando **svn propedit**—um arquivo temporário é preenchido com o valor atual da propriedade que o usuário pretende editar, e permite que o usuário realize as modificações neste programa editor (veja “Propriedades”). O valor padrão para esta opção é vazio. A ordem de prioridade para determinar o comando editor (em ordem crescente de precedência) é:

1. A opção de linha de comando `--editor-cmd`
2. A variável de ambiente `SVN_EDITOR`
3. A opção de configuração `editor-cmd`
4. A variável de ambiente `VISUAL`
5. A variável de ambiente `EDITOR`
6. Possivelmente, um valor padrão já embutido no Subversion (ausente em distribuições oficiais do Subversion)

O valor de quaisquer dessas opções ou variáveis (diferentemente do `diff-cmd`) é o começo de uma linha de comando a ser executada pelo shell. O Subversion concatena um espaço a o caminho do arquivo temporário a ser editado. O editor deve modificar o arquivo temporário e retornar um código zero como saída para indicar sucesso.

`diff-cmd`
: Isto especifica o caminho absoluto para um programa de diferenciação, usado quando o Subversion gera saída em formato “diff” (como ao usar o comando **svn diff**). Por padrão, o Subversion usa uma biblioteca interna de diferenciação—atribuir um valor a esta opção irá fazer com que o Subversion realize essa tarefa executando um programa externo. Veja “Usando Ferramentas Externas de Diferenciação” para mais detalhes sobre como usar tais programas.

`diff3-cmd`
: Isto especifica o caminho absoluto para um programa de diferenciação em três vias. O Subversion usa este programa para mesclar alterações feitas pelo usuário com aquelas recebidas do repositório. Por padrão, o Subversion usa uma biblioteca interna de diferenciação—atribuir um valor a esta opção irá fazer com que o Subversion execute esta tarefa usando um programa externo. Veja “Usando Ferramentas Externas de Diferenciação” para mais detalhes sobre como usar tais programas.

`diff3-has-program-arg`
: Esta opção deve ser atribuída para `true` se o programa especificado pela opção `diff3-cmd` aceitar um parâmetro `--diff-program` de linha de comando.

A seção `tunnels` permite que você defina um novo esquema de túnel para uso com o **svnserve** e clientes de conexões `svn://`. Para mais detalhes, veja “Tunelamento sobre SSH”.

A seção `miscellany` é onde se encontram todas aquelas opções que não pertençam a nenhuma das outras seções. [2] Nesta seção, você pode encontrar:

`global-ignores`
: Ao executar o comando **svn status**, o Subversion lista arquivos e diretórios não-versionados juntamente com os versionados, marcando-os com um caractere ? (veja “Obtendo uma visão geral de suas alterações”). Algumas vezes, pode ser chato ficar sempre vendo itens não-versionados, desinteressantes—por exemplo, arquivos objeto que resultantes da compilação de um programa —na tela. A opção `global-ignores` é uma lista de campos delimitados por espaços e que descrevem os nomes de arquivos e diretórios que o Subversion não deveria exibir a não ser

[2] Alguém para um jantar improvisado?

apenas quando versionados. Seu valor padrão é `*.o *.lo *.la #*# .*.rej *.rej .*~ *~ .#* .DS_Store`.

Bem como o **svn status**, os comandos **svn add** e **svn import** também ignoram arquivos que correspondam com esta lista quando estão escaneando um diretório. Você pode sobrescrever este comportamento para uma única instância de qualquer desses comandos especificando explicitamente o nome do arquivo ou usando a opção `--no-ignore` em linha de comando.

Para mais informação sobre controle granularizado de itens ignorados, veja “Ignorando Itens Não-Versionados”.

`enable-auto-props`
: Isto faz com que o Subversion defina propriedades automaticamente em novos arquivos adicionados ou importados. Seu valor padrão é `no`, altere-o para `yes` para habilitar propriedades automáticas. A seção `auto-props` deste arquivo especifica quais propriedades serão definidas em quais arquivos.

`log-encoding`
: Esta variável define a codificação de caracteres padrão para mensagens de log de submissões. É uma versão permanente da opção `--encoding` (veja “Opções do svn”). O repositório Subversion armazena mensagens de log em UTF-8, e assume que suas mensagens de log sejam escritas usando a localização nativa do sistema operacional. Você deve especificar uma codificação diferente se suas mensagens de commit forem escritas em um outro conjunto de caracteres.

`use-commit-times`
: Normalmente, os arquivos de sua cópia de trabalho têm registros de tempo que refletem o momento da última vez em que eles foram referenciados ou alterados por algum processo, que pode ser seu próprio editor ou mesmo algum subcomando do **svn**. Isto geralmente é conveniente para pessoas que desenvolvem software, porque sistemas de compilação quase sempre verificam registros de tempo como uma forma de decidir quais arquivos precisam ser recompilados.

Em outras situações, porém, algumas vezes é bom ter registros de tempo de arquivos na cópia de trabalho que reflitam a última vez em que eles foram modificados no repositório. O comando **svn export** sempre põe estes “registros de tempo do último commit” nas árvores geradas por ele. Ao definir esta variável de configuração para `yes`, os comandos **svn checkout**, **svn update**, **svn switch**, e **svn revert** passarão a registrar momentos de tempo dos comandos de último commit nos arquivos que manipularem.

A seção `auto-props` controla a capacidade de um cliente Subversion de atribuir propriedades automaticamente em arquivos quando eles forem adicionados ou importados. Ela contém um conjunto de pares chave-valor no formato `PATTERN = PROPNAME=PROPVALUE` onde `PATTERN` é um padrão que corresponde a um conjunto de nomes de arquivo e o restante da linha é a propriedade e seu valor. Múltiplas correspondências para um arquivo irão resultar em múltiplas propriedades definidas para aquele arquivo; no entanto, não há garantia de que essas propriedades automáticas serão aplicadas na ordem na qual estiverem listadas no arquivo de configuração, então você não pode fazer com que uma regra “sobrescreva”. Você pode encontrar diversos exemplos do uso de propriedades automáticas no arquivo `config`. E por último, não se esqueça de atribuir `enable-auto-props` para `yes` na seção `miscellany` se você quiser habilitar propriedades automáticas.

Localização

Localização é o ato de fazer com que programas comportem-se de um modo específico de região. Quando um programa formata números ou datas em um modo específico para sua parte do mundo, ou imprime mensagens (ou aceita uma entrada) em sua linguagem nativa, o programa é dito ser *localizado*. Esta seção descreve os passos que o Subversion tem feito a respeito de localização.

Compreendendo localidades

A maioria dos sistemas operacionais modernos têm uma noção da “localidade atual”—isto é, a região ou país cujas convenções de localização são aplicadas. Estas convenções—normalmente escolhidas por

algum mecanismo de configuração de execução no computador—afetam a forma como os programas apresentam dados para o usuário, bem como a forma como eles aceitam dados fornecidos pelo usuário.

Na maioria dos sistemas Unix, você pode verificar os valores das opções de configuração relacionadas à localidade pela execução do comando **locale**:

```
$ locale
LANG=
LC_COLLATE="C"
LC_CTYPE="C"
LC_MESSAGES="C"
LC_MONETARY="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_ALL="C"
```

A saída é uma lista de variáveis de ambiente, relacionadas com a localidade, e seus valores atuais. Neste exemplo, as variáveis estão todas definidas com o local padrão `C`, mas os usuários podem definir estas variáveis para especificar combinações de código país/idioma. Por exemplo, se alguém quisesse definir a variável `LC_TIME` com o valor `fr_CA`, então os programas saberiam apresentar informações de data e hora formatadas de acordo com a expectativa de um Canadense que fala Francês. E se alguém quisesse definir a variável `LC_MESSAGES` com o valor `zh_TW`, então os programas saberiam apresentar mensagens legíveis em Chinês Tradicional. Configurar a variável `LC_ALL` tem o efeito de modificar todas as variáveis de localidade para o mesmo valor. O valor de `LANG` é utilizado como um valor padrão para qualquer variável de localidade que está indefinida. Para ver a lista de localidades disponíveis em um sistema Unix, execute o comando **locale -a**.

No Windows, a configuração de localidade é feita por meio do item "Opções Regionais e de Idioma" no painel de controle. Lá você poderá ver e selecionar os valores das configurações individuais das localidades disponíveis, e até mesmo personalizar (a um nível de detalhe enjoativo) várias das convenções de formatação de exibição.

Uso de localidades do Subversion

O cliente Subversion, **svn**, aplica a configuração de localidade atual de duas formas. Primeiro, ele trata o valor da variável `LC_MESSAGES` e tenta imprimir todas as mensagens no idioma especificado. Por exemplo:

```
$ export LC_MESSAGES=de_DE
$ svn help cat
cat: Gibt den Inhalt der angegebenen Dateien oder URLs aus.
Aufruf: cat ZIEL[@REV]...
…
```

Este comportamento ocorre identicamente tanto em sistemas Unix quando em Windows. Note, entretanto, que enquanto seu sistema operacional possa ter suporte para uma certa localidade, o cliente do Subversion ainda pode não estar hábil para falar com o idioma específico. A fim de produzir mensagens localizadas, pessoas voluntárias podem fornecer traduções para cada linguagem. As traduções são escritas usando o pacote GNU gettext, que resultam em módulos de tradução que terminam com a extensão `.mo` no nome do arquivo. Por exemplo, o arquivo de tradução para o Alemão é nomeado `de.mo`. Estes arquivos de tradução estão instalados em algum lugar em seu sistema. No Unix, eles normalmente ficam em `/usr/share/locale/`, enquanto que no Windows eles são frequentemente encontrados na pasta `\share\locale\` na área de instalação do Subversion. Uma vez instalado, um módulo que contém o nome do programa que fornece as traduções. Por exemplo, o arquivo `de.mo` pode vir a ser finalmente instalado como `/usr/share/locale/de/LC_MESSAGES/subversion.mo`. Ao navegar pelos arquivos `.mo` instalados, você pode ver quais linguagens o cliente Subversion está hábil para falar.

A segunda forma na qual a localidade é aplicada envolve como o **svn** interpreta suas entradas. O repositório armazena todos os caminhos, nomes de arquivo, e mensagens de log em Unicode, codificadas como UTF-8. Nesse sentido, o repositório está *internacionalizado*—isto é, o repositório está pronto para aceitar entradas em qualquer linguagem humana. Isso significa, contudo, que o cliente do Subversion é responsável por enviar somente nomes de arquivos e mensagens de log em UTF-8 para o repositório. Para isso, ele deve converter os dados a partir da localidade nativa em UTF-8.

Por exemplo, suponha que você criou uma arquivo chamado `caffÃ¨.txt`, e então ao submeter o arquivo, você escreve a mensagem de log como "Adesso il caffÃ¨ Ã¨ piÃ¹ forte". Tanto o nome do arquivo quanto a mensagem de log contem caracteres não-ASCII, porém, sua localidade está definida como `it_IT`, e o cliente Subversion sabe interpretá-los como Italiano. Ele usa um conjunto de caracteres do Italiano para converter os dados para UTF-8 antes de enviá-los para o repositório.

Observe que, enquanto o repositório demanda nomes de arquivo e mensagens de log em UTF-8, ele *não* presta atenção ao conteúdo do arquivo. O Subversion trata o conteúdo do arquivo como sequências de bytes quaisquer, e nem o cliente nem servidor fazem uma tentativa de entender o conjunto de caracteres ou codificação do conteúdo.

Erros de conversão de conjuntos de caractere

Quando estiver usando o Subversion, você pode receber um erro relacionado a conversões de conjuntos de caractere:

```
svn: Can't convert string from native encoding to 'UTF-8':
…
svn: Can't convert string from 'UTF-8' to native encoding:
…
```

Erros como estes normalmente ocorrem quando o cliente do Subversion recebeu uma sequência UTF-8 do repositório, mas nem todos os caracteres nesta sequência podem ser representados usando a codificação da localidade atual. Por exemplo, se sua localidade é `en_US` mas um colaborador submeteu um nome de arquivo em Japonês, você provavelmente verá este erro quando for receber o arquivo durante um **svn update**.

A solução está tanto em definir sua localidade para alguma que *possa* representar os dados UTF-8 recebidos, ou modificar o nome do arquivo ou mensagem de log dentro do repositório. (E não esqueça de dar uma bofetada na mão de seu colaborador—os projetos devem decidir o quanto antes as linguagens comuns, para que todos participantes estejam utilizando a mesma localidade.)

Usando Ferramentas Externas de Diferenciação

A presença das opções `--diff-cmd` e `--diff3-cmd`, e dos parâmetros de configuração em tempo de execução similarmente nomeados (veja "Configuração"), podem levar a uma falsa noção do qual fácil seja usar ferramentas externas de diferenciação (ou "diff") ou de fusão com o Subversion. Ainda que o Subversion possa usar a maioria das ferramentas populares disponíveis, o esforço investido nesta configuração quase sempre acaba por se tornar algo não-trivial.

A interface entre o Subversion e ferramentas externas de diferenciação e fusão remontam a uma época em que as únicas capacidades de diferenciação do Subversion eram construídas sobre invocações de utilitários do kit de ferramentas diffutils do GNU, especificamente os utilitários **diff** e **diff3**. Para conseguir o tipo de comportamento de que o Subversion precisa, ele chamava esses utilitários com mais algumas opções e parâmetros úteis, muitas das quais eram bastante específicas desses utilitários. Algum tempo depois, o Subversion desenvolveu-se com sua própria biblioteca de diferenciação, e com

mecanismos de recuperação de falhas, [3] as opções `--diff-cmd` e `--diff3-cmd` foram adicionadas ao cliente de linha de comando do Subversion e então os usuários puderam indicar com mais facilidade se preferiam usar os utilitários GNU diff e diff3 ao invés da nova biblioteca biblioteca interna de diferenciação. Se essas opções forem usadas, o Subversion deverá simplesmente ignorar sua biblioteca interna de diferenciação, voltando a usar programas externos, longas listas de argumentos e tudo o mais. E é assim que as coisas permanecem até hoje.

Não demora muito para as pessoas perceberem que dispor de tais mecanismos de configuração para especificar que o Subversion deve usar os utilitários GNU diff e diff3 localizados em um local específico no sistema poderia ser aplicado também para o uso de outras ferramentas de diferenciação e fusão. No fim das contas, o Subversion atualmente nem verifica se as coisas que mencionamos executar eram membros do kit de ferramentas GNU diffutils. Mas o único aspecto configurável ao usar estas ferramentas externas é a sua localização no sistema—e não o conjunto de opções, parâmetros, ordem, etc. O Subversion continua incluindo todas essas opções de utilitários GNU à sua ferramenta externa de diferenciação independentemente se o programa em questão pode entender tais opções ou não. E é aqui que as coisas deixam de ser intuitivas para a maioria dos usuários.

O ponto chave ao usar ferramentas externas de diferenciação e fusão com o Subversion (que não as diff e diff3 do kit GNU, é claro) é usar scripts encapsuladores que convertam a entrada do Subversion em algo que sua ferramenta de diferenciação possa entender, e então converter a saída de sua ferramenta de volta num formato esperado pelo Subversion—o formato que as ferramentas GNU deveriam usar. As seções seguintes abordam os detalhes sobre esses formatos.

A decisão sobre quando disparar uma diferenciação ou fusão contextual como parte de uma operação maior do Subversion é feita inteiramente pelo Subversion, e é afetada por, entre outras coisas, se os arquivos em questão sobre os quais o sistema estiver operando estiver ou não em um formato legível por humanos como determinado a partir de sua propriedade `svn:mime-type`. Isto quer dizer, por exemplo, que mesmo que se você tiver a ferramenta ou plugin de diferenciação ou fusão do Microsoft Word mais esperta do universo, ela nunca deveria ser invocada pelo Subversion se seus documentos do Word sob controle de versão possuírem um tipo MIME configurado que denote que eles não são legíveis por humanos (tal como `application/msword`). Para mais informações sobre configurações de tipos MIME, veja “Tipo de Conteúdo do Arquivo”

Ferramentas diff Externas

O Subversion chama programas de diferenciação externos com os parâmetros que são adequados para o utilitário GNU diff, e espera apenas que o programa externo retorne para o sistema com um código indicando sucesso. Para a maioria dos programas diff alternativos, apenas o sexto e sétimo argumentos —os caminhos dos arquivos que representam os lados esquerdo e direito do diff, respectivamente— são relevantes. Note que o Subversion executa o programa diff uma vez para os arquivos modificados abordados por uma operação do Subversion, então se seu programa executa de maneira assíncrona (ou em “segundo plano”), você pode ter diversas instâncias todas executando simultaneamente. E finalmente, o Subversion espera que seu programa retorne um código de erro igual a 1 se seu programa tiver encontrado quaisquer diferenças, ou 0 em caso contrário—qualquer outro código resultante é considerado como um erro fatal. [4]

Exemplo 7.2, “diffwrap.sh” e Exemplo 7.3, “diffwrap.bat” são modelos para scripts encapsuladores para ferramentas diff externas, em formato de arquivos shell Bourne e batch do Windows, respectivamente.

[3] Os desenvolvedores do Subversion são muito bons, mas até os melhores às vezes comentem erros.

[4] A página de manual do GNU diff descreve isso da seguinte forma: “Uma saída de status de 0 significa que não foram encontradas diferenças, 1 significa que algumas diferenças foram encontradas, e 2 significa problemas.”

Exemplo 7.2. diffwrap.sh

```
#!/bin/sh

# Configura seu programa diff preferido aqui.
DIFF="/usr/local/bin/my-diff-tool"

# O Subversion dispõe dos caminhos que precisamos como o sexto e
# sétimo parâmetros.
LEFT=${6}
RIGHT=${7}

# Chama o comando diff (modifique a linha a seguir de acordo com seu
# programa).
$DIFF --left $LEFT --right $RIGHT

# Retorna um código de erro de 0 se nenhumas diferenças forem
# encontradas, ou 1 em caso contrário.  Qualquer outro código de erro
# será considerado como fatal.
```

Exemplo 7.3. diffwrap.bat

```
@ECHO OFF

REM Configura seu programa diff preferido aqui.
SET DIFF="C:\Program Files\Funky Stuff\My Diff Tool.exe"

REM O Subversion dispõe dos caminhos que precisamos como o sexto e
REM sétimo parâmetros.
SET LEFT=%6
SET RIGHT=%7

REM Chama o comando diff (modifique a linha a seguir de acordo com seu
REM programa).
%DIFF% --left %LEFT% --right %RIGHT%

REM Retorna um código de erro de 0 se nenhumas diferenças forem
REM encontradas, ou 1 em caso contrário.  Qualquer outro código de erro
REM será considerado como fatal.
```

Ferramentas diff3 Externas

O Subversion chama programas externos de fusão de texto com parâmetros adequados para o utilitário GNU diff3, esperando que o programa externo retorne com para o sistema com um código indicando sucesso e que o conteúdo completo do arquivo resultante da fusão de texto seja exibido na saída padrão (para que então o Subversion possa redirecioná-la adequadamente para o arquivo sob controle de versão). Para a maioria dos programas de fusão alternativos, apenas o nono, décimo e décimo primeiro argumentos—os caminhos dos arquivos que representam a “minha” versão do arquivo, a “mais antiga” e a “sua” versão do arquivo, respectivamente—são relevantes. Note que como o Subversion depende da saída de seu programa de fusão de texto, seu script de encapsulamento não deve encerrar antes que a saída tenha sido repassada ao Subversion. Quando o script finalmente terminar, ele deveria retornar um código de status de 0 se a fusão foi executada com sucesso, ou 1 se conflitos ainda permanecerem na saída—qualquer outro código de erro é considerado como erro fatal.

Exemplo 7.4, “diff3wrap.sh” e Exemplo 7.5, “diff3wrap.bat” são modelos para scripts encapsuladores para ferramentas de fusão de texto externas, em formato de arquivos shell Bourne e batch do Windows, respectivamente.

Exemplo 7.4. diff3wrap.sh

```
#!/bin/sh

# Configura seu programa diff3 preferido aqui.
DIFF3="/usr/local/bin/my-merge-tool"

# O Subversion dispõe dos caminhos que precisamos como o nono, décimo e
# décimo primeiro parâmetros.
MINE=${9}
OLDER=${10}
YOURS=${11}

# Chama o comando de fusão de texto (modifique a linha a seguir de
# acordo com seu programa).
$DIFF3 --older $OLDER --mine $MINE --yours $YOURS

# Depois de executar a fusão de texto, este script precisa imprimir o
# conteúdo do arquivo resultante para stdout.  Você pode fazer isso da
# forma que achar melhor.  Retorna um código de erro de 0 no caso de uma
# fusão de sucesso, 1 se conflitos não resolvidos ainda permanecerem no
# arquivo resultante.  Qualquer outro código de erro será tratado como
# fatal.
```

Exemplo 7.5. diff3wrap.bat

```
@ECHO OFF

REM Configura seu programa diff3 preferido aqui.
SET DIFF3="C:\Program Files\Funky Stuff\My Merge Tool.exe"

REM O Subversion dispõe dos caminhos que precisamos como o nono, décimo e
REM décimo primeiro parâmetros.  Mas só temos acesso a nove parâmetros de
REM cada vez, então deslocamos nosso nove parâmetros duas vezes para que
REM possamos obter os valores que precisamos.
SHIFT
SHIFT
SET MINE=%7
SET OLDER=%8
SET YOURS=%9

REM Chama o comando de fusão de texto (modifique a linha a seguir de
REM acordo com seu programa).
%DIFF3% --older %OLDER% --mine %MINE% --yours %YOURS%

REM Depois de executar a fusão de texto, este script precisa imprimir o
REM conteúdo do arquivo resultante para stdout.  Você pode fazer isso da
REM forma que achar melhor.  Retorna um código de erro de 0 no caso de uma
REM fusão de sucesso, 1 se conflitos não resolvidos ainda permanecerem no
REM arquivo resultante.  Qualquer outro código de erro será tratado como
REM fatal.
```

Capítulo 8. Incorporando o Subversion

O Subversion tem uma estrutura modular: ele é implementado como uma coleção de bibliotecas escritas em C. Cada biblioteca tem um propósito bem definido e uma Interface de Programação de Aplicação (API - Application Programming Interface), e esta interface está disponível não só para o próprio Subversion usar, mas para qualquer software que queira incorporar ou controlar o Subversion através de programação. Adicionalmente, a API do Subversion está disponível não só para outros programas em C, mas também para programas em linguagens de alto nível como Python, Perl, Java ou Ruby.

Este capítulo é para aqueles que desejam interagir com o Subversion através da sua API pública ou seus vários vínculos de linguagem. Se você deseja escrever scripts robustos ao redor das funcionalidades do Subversion para simplificar sua vida, se está tentando desenvolver integrações mais complexas entre o Subversion e outras partes de um software, ou apenas tem interesse nas várias bibliotecas modulares e o que elas tem a oferecer, este capítulo é para você. Se, entretanto, você não se vê participando com o Subversion nesse nível, sinta-se livre para pular este capítulo certo que suas experiências como usuário do Subversion não serão afetadas.

Projeto da Biblioteca em Camadas

Cada uma das bibliotecas centrais do Subversion podem existir em uma de três principais camadas —na Camada do Repositório, na Camada de Acesso ao Repositório (RA), ou na Camada Cliente (veja Figura 1, “Arquitetura do Subversion”). Nós examinaremos essas camadas daqui a pouco, mas primeiro, vamos brevemente dar uma olhada nas várias bibliotecas do Subversion. Pelo bem da consistência, nós referiremos às bibliotecas pelos seus nomes de biblioteca Unix sem extensão (libsvn_fs, libsvn_wc, mod_dav_svn, etc.).

libsvn_client
: Interface primária para programas cliente

libsvn_delta
: Rotinas de diferenciação de árvore e fluxo de bytes

libsvn_diff
: Rotinas de diferenciação e mesclagem contextuais

libsvn_fs
: Base comum e carregador de módulo para sistema de arquivos

libsvn_fs_base
: A base para sistema de arquivos do Berkeley DB

libsvn_fs_fs
: A base para sistema de arquivos nativo (FSFS)

libsvn_ra
: Base comum e carregador de módulo para Acesso ao Repositório

libsvn_ra_dav
: O módulo para Acesso ao Repositório por WebDAV

libsvn_ra_local
: O módulo de Acesso ao Repositório localmente

libsvn_ra_serf
: Outro módulo (experimental) para Acesso ao Repositório por WebDAV

libsvn_ra_svn
: O módulo de Acesso ao Repositório do protocolo padrão

libsvn_repos
: Interface de Repositório

libsvn_subr
: Diversas sub-rotinas úteis

libsvn_wc
: A biblioteca de gerenciamento de cópia de trabalho

mod_authz_svn
: Módulo de autorização do Apache para acesso a repositórios do Subversion via WebDAV

mod_dav_svn
: Módulo Apache para mapeamento de operações WebDAV para operações do Subversion

O fato da palavra “diversas” só aparecer uma vez na lista anterior é um bom sinal. A equipe de desenvolvimento do Subversion se esforça bastante para fazer com que as funcionalidades fiquem na camada certa e nas bibliotecas certas. Talvez a maior vantagem de um sistema modular é a pouca complexidade do ponto de vista do desenvolvedor. Como desenvolvedor, você pode enxergar rapidamente um tipo de “grande figura” que permite você achar o local de certos pedaços de funcionalidade com certa facilidade.

Outro benefício da modularidade é a habilidade de substituir um dado módulo por uma biblioteca inteiramente nova que implementa a mesma API sem afetar o resto do código base. De certo modo, isso já acontece dentro do Subversion. Cada uma das bibliotecas libsvn_ra_dav, libsvn_ra_local, libsvn_ra_serf, e libsvn_ra_svn implementam a mesma interface, todas funcionando como extensões de libsvn_ra. E todas as quatro se comunicam com a Camada do Repositório—libsvn_ra_local conecta-se ao repositório diretamente; as outras três através de uma rede. As bibliotecas libsvn_fs_base e libsvn_fs_fs são outro par de bibliotecas que implementam a mesma funcionalidade de maneiras diferentes—ambas são extensões para a biblioteca libsvn_fs comum.

O cliente também mostra os benefícios da modularidade no projeto do Subversion. A biblioteca lbsvn_client do Subversion é uma boa para a maioria das funcionalidades necessárias para projetar um útil cliente do Subversion (veja “Camada Cliente”). Portanto, embora a distribuição do Subversion ofereça apenas o programa de linha de comando **svn** como cliente, existem várias ferramentas de terceiros que oferecem várias formas de interface gráfica com o usuário. Essas interfaces gráficas usam a mesma API que o cliente de linha de comando usa. Este tipo de modularidade tem tido um papel importante na proliferação de clientes Subversion acessíveis e integrações em IDEs e, por consequência, pela tremenda taxa de adoção do próprio Subversion.

Camada de Repositório

Quando referirmos à Camada de Repositório do Subversion, estamos geralmente falando sobre dois conceitos básicos—a implementação de sistema de arquivos versionados (acessada via libsvn_fs, e suportada por suas extensões libsvn_fs_base e libsvn_fs_fs), e a lógica do repositório que a encapsula (implementada em libsvn_respos). Estas bibliotecas provem os mecanismos de armazenamento e relatórios para as várias revisões de seus dados em controle de versão. Esta camada está conectada com a Camada de Cliente através da Camada de Acesso ao Repositório, e é, da da perspectiva do usuário Subversion, a coisa no “outro extremo da linha”.

O Sistema de Arquivos Subversion não é um sistema de arquivos a nível de kernel que se pode instalar em um sistema operacional (como o ext2 do Linux ou o NTFS), mas um sistema de arquivos virtual. Além de armazenar “arquivos” e “diretórios” como arquivos e diretórios reais (como os que você pode navegar usando seu programa de terminal favorito), ele usa um dos dois servidores abstratos de armazenamento disponíveis—seja um ambiente do banco de dados Berkeley DB, ou uma representação em arquivos planos. (Para aprender mais sobre os dois servidores de repositório, veja “Escolhendo uma Base de Dados”.) Houve até um considerável interesse na comunidade de desenvolvimento em dar a futuras liberações do Subversion a habilidade de usar outros sistemas de base de dados, talvez através de um mecanismo como um ODBC (Open Database Connectivity). De fato, a Google fez algo similar a isto antes de lançar o serviço Google Code para hospedagem de

projetos: eles anunciaram em meados de 2006 que os membros de sua equipe Open Source teriam escrito uma nova extensão proprietária para o sistema de arquivos do Subversion que usaria seu banco de dados ultra-escalável Bigtable em seu armazenamento.

A API de sistema de arquivos exportada por libsvn_fs contém os tipos de funcionalidade que você esperaria de qualquer outra API de sistema de arquivos—você pode criar e remover arquivos e diretórios, copiar e movê-los, modificar o conteúdo de arquivos, e assim por diante. Ela também tem funcionalidades que não são tão comuns, como a habilidade de adicionar, modificar, e remover metadados (“propriedades”) em cada arquivo ou diretório. Além de tudo, o Sistema de Arquivos Subversion é um sistema de versionamento, o que significa que assim que você faz mudanças em sua árvore de diretórios, o Subversion lembra como sua árvore estava quando essas mudanças ocorreram. E as mudanças anteriores. E as anteriores a essas. E assim por diante, todo o caminho de volta durante o tempo de versionamento até (e no máximo) o primeiro momento em que você iniciou a adição de coisas ao sistema de arquivos.

Todas as modificações que você fez em sua árvore são feitas dentro do contexto de uma transação de submissão do Subversion. A seguir veremos uma rotina simplificada e genérica de modificação de seu sistema de arquivos:

1. Iniciar uma transação de submissão do Subversion.
2. Fazer suas mudanças (adicionar, excluir, modificações de propriedades, etc.).
3. Submeter sua transação.

Uma vez que você submeteu sua transação, suas modificações no sistema de arquivo são permanentemente armazenadas como artefatos de histórico. Cada um desses ciclos produz uma única e nova revisão de sua árvore, e cada revisão será sempre acessível como uma imagem imutável de “como as coisas estavam”.

A Confusão com Transações

A noção de transação no Subversion pode ser facilmente confundida com o suporte a transações oferecido pelo banco de dados encapsulado propriamente dito, especialmente pelo código de libsvn_fs_base se aproximar ao código do banco de dados Berkeley DB. Ambos os tipos de transação existem para prover atomicidade e isolamento. Em outra palavras, transações dão a você a habilidade de realizar um conjunto de ações em uma forma “tudo ou nada”—ou todas as ações são completadas com sucesso, ou todas são tratadas como se *nenhuma* delas tivesse ocorrido—e em modo que não exista interferência com outros processos atuando nos dados.

As transações de banco de dados geralmente inclui pequenas operações relacionadas especificamente com a modificações de dados no banco de dados propriamente dito (como modificar o conteúdo de uma linha em uma tabela). As transações do Subversion são maiores em escopo, incluindo operações de alto nível, como fazer modificações em um conjunto de arquivos e diretórios os quais serão armazenados como a próxima revisão da árvore do sistema de arquivos. Como se já não fosse confuso o suficiente, considere o fato de que o Subversion usa uma transação de banco de dados durante a criação de uma transação Subversion (de modo que, se a criação da transação do Subversion falhar, o banco de dados ficará como se nunca tivéssemos tentado esta criação no primeiro local)!

Felizmente para os usuários da API de sistema de arquivos, o suporte a transação provido pelo sistema de banco de dados está quase totalmente escondido para exibição (como seria esperado de um esquema de bibliotecas devidamente modularizadas). Apenas quando você começa a procurar dentro da implementação do próprio sistema de arquivos que essas coisas começam a ficar visíveis (ou interessantes).

A maioria das funcionalidades providas pela interface de sistema de arquivos lida com ações que ocorrem em caminhos de sistema de arquivos individuais. Isto é, de fora do sistema de arquivos, o mecanismo primário para descrever e acessar as revisões individuais de arquivos e diretórios vem através do uso de de caminhos como `/foo/bar`, como se você estivesse endereçando arquivos e diretórios através de seu programa de terminal favorito. Você adiciona novos arquivos e diretórios

passando seus futuros caminhos para as funções certas da API. Você requisita informações sobre eles pelo mesmo mecanismo.

Ao contrário de muitos sistemas de arquivos, entretanto, um caminho sozinho não é informação suficiente para identificar um arquivo ou diretório no Subversion. Pense na árvore de diretórios como um sistema de duas dimensões, onde um irmão de um nó representa uma espécie de movimento horizontal, e descendo nos subdiretórios um movimento vertical. Figura 8.1, “Arquivos e diretórios em duas dimensões” mostra uma típica representação de uma árvore exatamente assim.

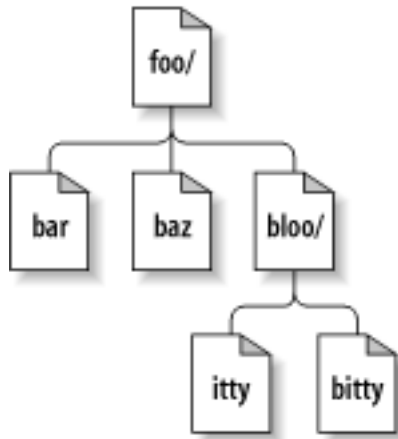
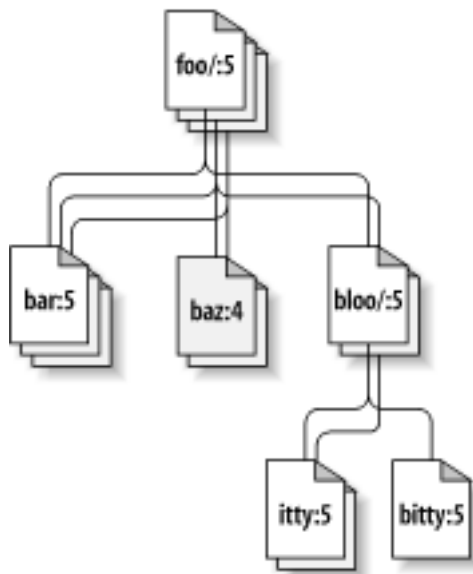
Figura 8.1. Arquivos e diretórios em duas dimensões

A diferença aqui é que o sistema de arquivos Subversion tem uma brilhante terceira dimensão que muitos sistemas de arquivos não possuem—Tempo! [1] Na interface de sistema de arquivos, quase toda função que tem um um argumento *`path`* também espera um argumento *`root`*. Este argumento svn_fs_root_t descreve tanto uma revisão quanto uma transação do Subversion (que é simplesmente uma revisão em formação), e oferece esse contexto de terceira dimensão necessário para entender a diferença entre `/foot/bar` na revisão 32, e o mesmo caminho existente na revisão 98. A Figura 8.2, “Versionando o tempo—a terceira dimensão!” mostra o histórico de revisões como uma dimensão adicional no universo do sistema de arquivos Subversion.

Figura 8.2. Versionando o tempo—a terceira dimensão!

[1] Nós entendemos que isso pode ser um choque para os fãs de ficção científica que estão a muito tempo sob a impressão de que Tempo é na verdade a *quarta* dimensão, e nós pedimos desculpamos por qualquer trauma emocional causado por nossa declaração de uma teoria diferente.

Como mencionado anteriormente, a API de libsvn_fs parece como qualquer outro sistema de arquivos, exceto que ele tem essa maravilhosa capacidade de versionamento. Ela foi projetada para ser usável por qualquer programa interessado em um sistema de arquivos com versionamento. Não coincidentemente, o próprio Subversion tem interesse nessa funcionalidade. Mas enquanto a API do sistema de arquivos deveria ser suficiente para suporte básico em versionamento de arquivos e diretórios, o Subversion quer mais—e é aí que aparece a libsvn_repos.

A biblioteca de repositório Subversion (libsvn_repos) está sentada (logicamente falando) no topo da API de libsvn_fs, provendo funcionalidades adicionais além daquelas essenciais à lógica do sistema de arquivos versionado. Ela não cobre completamente todas as funções de um sistema de arquivos—apenas alguns passos maiores no ciclo geral de atividade do sistema de arquivos são cobertas pela interface de repositório. Algumas dessas incluem a criação e submissão de transações Subversion, e a modificação de propriedades de revisão. Esses eventos particulares são cobertos pela na camada de repositório porque possuem ganchos associados a eles. Um sistema de ganchos de repositório não está estritamente relacionado à implementação de um sistema de arquivos com versionamento, então ele fica na biblioteca de cobertura do repositório

Não obstante, o mecanismo de ganchos é uma das razões para a abstração de uma biblioteca de repositório separada do restante do código do sistema de arquivo. A API de libsvn_repos oferece muitas outras utilidades importantes ao Subversion. Isso inclui as habilidades para:

- criar, abrir, destruir, e recuperar passos em um repositório Subversion e o sistema de arquivos incluído nesse repositório.
- descrever as diferenças entre duas árvores de sistema de arquivos.
- consultar as mensagens de log das submissões associadas com todas (ou algumas) das revisões nas quais um conjunto de arquivos foi modificado no sistema de arquivos.
- gerar um “despejo” do sistema de arquivos que seja legível ao ser humano, uma completa representação das revisões no sistema de arquivos.
- analisar este formato de despejo, carregando as revisões despejadas dentro de um repositório Subversion diferente.

Como o Subversion continua a evoluir, a biblioteca de repositório repositório crescerá com a biblioteca de sistema de arquivos para oferecer um número crescente de funcionalidades e opções de configuração.

Camada de Acesso ao Repositório

Se a Camada de Repositório do Subversion está “no outro extremo da linha”, a Camada de Acesso ao Repositório (RA) é a própria linha. Encarregado de guiar dados entre as bibliotecas clientes e o repositório, esta camada inclui a biblioteca libsvn_ra, módulo carregador, os próprios módulos RA (que normalmente incluem libsvn_ra_dav, libsvn_ra_local, libsvn_ra_serf, e libsvn_ra_svn), e qualquer biblioteca adicional necessária para um ou mais desses módulos RA (assim como o módulo Apache mod_dav_svn ou o servidor de libsvn_ra_svn, **svnserv**).

Já que Subversion usa URLs para identificar suas fontes de repositório, a parte protocolo do esquema URL (normalmente `file://`, `http://`, `https://`, `svn://`, ou ou `svn+ssh://`) é usada para determinar qual módulo RA manipulará as comunicações. Cada módulo registra uma lista dos protocolos que sabem como “conversar” para que o carregador de RA possa, em tempo de execução, determinar qual módulo usar para realizar a tarefa. Você pode descobrir quais módulos RA estão disponíveis ao cliente de linha de comando do Subversion e que protocolos eles suportam, executando **svn --version**:

```
$ svn --version
svn, version 1.4.3 (r23084)
   compiled Jan 18 2007, 07:47:40
```

```
Copyright (C) 2000-2006 CollabNet.
Subversion is open source software, see http://subversion.tigris.org/
This product includes software developed by CollabNet (http://www.Collab.Net/).

The following repository access (RA) modules are available:

* ra_dav : Module for accessing a repository via WebDAV (DeltaV) protocol.
  - handles 'http' scheme
  - handles 'https' scheme
* ra_svn : Module for accessing a repository using the svn network protocol.
  - handles 'svn' scheme
* ra_local : Module for accessing a repository on local disk.
  - handles 'file' scheme

$
```

A API pública exportada pela Camada RA contém as funcionalidades necessárias para enviar e receber dados versionados para e do repositório. E cada uma das extensões disponíveis está hábil para realizar esta tarefa usando um protocolo específico—libsvn_ra_dav conversa em HTTP/WebDAV (opcionalmente usando criptografia com SSL) com um Servidor HTTP Apache que estiver rodando o módulo mod_dav_svn de servidor Subversion; libsvn_ra_svn conversa num protocolo de rede especial com o programa **svnserve**; e assim por diante.

E para aqueles que desejam acessar um repositório Subversion usando ainda outro protocolo, isso é perfeitamente possível porque a Camada de Acesso ao Repositório é modularizada! Desenvolvedores podem simplesmente escrever uma nova biblioteca que implementa a interface RA em um dos lados e comunicar com o repositório no outro lado. Sua nova biblioteca pode usar protocolos de rede já existentes, ou você pode inventar o seu próprio. Você poderia usar chamadas de comunicação entre processos (IPC), ou—vamos ficar loucos, devemos?—você pode até implementar um protocolo baseado em email. O Subversion oferece as APIs; você oferece a criatividade.

Camada Cliente

No lado do cliente, a cópia de trabalho do Subversion é onde todas as ações tomam lugar. A pilha de funcionalidades implementadas pelas bibliotecas do lado cliente existem para o único propósito de gerenciar cópias de trabalho—diretórios cheios de arquivos e outros sub-diretórios que servem como uma espécie de “reflexos” locais e editáveis de um ou mais locais do repositório—e propagando mudanças para e a partir da Camada de Acesso ao Repositório.

A biblioteca de cópia de trabalho do Subversion, libsvn_wc, é diretamente responsável por gerenciar os dados nas cópias de trabalho. Para conseguir isso, a biblioteca armazena informações administrativas sobre cada diretório da cópia de trabalho dentro de um sub-diretório especial. Este sub-diretório, nomeado `.svn`, está presente em cada diretório da cópia de trabalho e contém vários outros arquivos e diretórios que registram o estado e oferecem um espaço de trabalho privado para as ações de administração. Para os familiares com o CVS, este sub-diretório `.svn` é similar em propósito aos diretórios administrativos CVS encontrados em cópias de trabalho CVS. Para mais informações sobre a área administrativa `.svn`, veja “Por dentro da Área de Administração da Cópia de Trabalho” neste capítulo.

A biblioteca do cliente Subversion, libsvn_client, tem uma responsabilidade mais abrangente; seu trabalho é combinar a funcionalidade da biblioteca de cópia de trabalho com as da Camada de Acesso ao Repositório, e então oferecer uma API de alto nível para qualquer aplicação que desejar realizar ações gerais de controle de revisão. Por exemplo, a função `svn_client_checkout()` pega uma URL como argumento. Ela passa esta URL para a camada de RA e abre uma sessão autenticada com um determinado repositório. Ela então pede ao repositório por uma certa árvore, e envia esta árvore para a biblioteca de cópia de trabalho, que então escrever toda a cópia de trabalho no disco (o diretório `.svn` e tudo mais).

A biblioteca cliente é projetada para ser usada por qualquer aplicação. Enquanto o código fonte do Subversion inclui um cliente de linha de comando padrão, deveria ser muito fácil escrever qualquer número de clientes GUI sob essa biblioteca cliente. Novas interfaces gráficas (ou qualquer novo cliente) para o Subversion não precisam ser invólucros desajeitados sobre o cliente de linha de linha de comando—elas possuem acesso total pela API de libsvn_client às mesmas funcionalidades, dados e mecanismos de resposta que o cliente de linha de comando usa. De fato, a árvore do código fonte do Subversion contém um pequeno programa em C (o qual pode ser encontrado em `tools/examples/minimal_client.c` que exemplifica como manusear a API do Subversion para criar um simples programa cliente.

Vinculando Diretamente—Uma Palavra Sobre o que é Certo

Porque seu programa GUI deveria vincular-se diretamente com libsvn_client em vez de agir como uma cobertura em volta do programa de linha de comando? Além de ser mais eficiente, pode ser mais correto também. Um programa de linha de comando (como o oferecido com o Subversion) que se vincula com a biblioteca cliente precisa traduzir eficientemente os bits das respostas e requisições de dados dos tipos C para alguma forma de saída legível por seres humanos. Esse tipo de tradução pode ser dispendioso. Sendo assim, o programa pode não mostrar todas as informações colhidas pela API, ou pode combinar pedaços de informações para uma apresentação compacta.

Se você encobrir um programa de linha de comando com outro programa, o segundo programa terá acesso apenas às informações já interpretadas (e como mencionado, possivelmente incompletas), que devem ser *novamente* traduzidas para *seu próprio* formato de apresentação. Com cada camada de encapsulamento, a integridade dos dados originais é potencialmente degradada mais e mais, quase como o resultado de fazer uma cópia de uma cópia (de uma cópia …) do seu cassete de áudio ou vídeo favorito.

Mas o argumento mais contundente para vincular diretamente às APIs em vez de encapsular outros programas é que o projeto Subversion fez promessas de compatibilidade entre suas APIs. Através de versões menores dessas APIs (como entre 1.3 e 1.4), nenhum protótipo de função mudará. Em outras palavras, você não será forçado a atualizar o código fonte de seu programa simplesmente porque você atualizou para uma nova versão do Subversion. Algumas funções podem ficar defasadas, mas elas ainda funcionarão, e isso te dá um intervalo de tempo para começar a usar as novas APIs. Esse tipo de compatibilidade não é prometido para as mensagens de saída do cliente de linha de comando do Subversion, as quais estão sujeitas a mudanças de uma versão para outra.

Por dentro da Área de Administração da Cópia de Trabalho

Como mencionamos anteriormente, cada diretório de uma cópia de trabalho do Subversion contém um sub-diretório especial chamado `.svn` que hospeda dados administrativos sobre este diretório da cópia de trabalho. O Subversion usa a informação em `.svn` para rastrear coisas como:

- Quais locais do repositório são representados pelos arquivos e subdiretórios no diretório da cópia de trabalho.
- Que revisão de cada um destes arquivos e diretórios está atualmente presente na sua cópia de trabalho.
- Todas propriedades definidas pelo usuário que possam estar anexadas a esses arquivos e diretórios.
- Cópias intactas (não alteradas) de arquivos da cópia de trabalho.

A estrutura e o conteúdo da área de administração da cópia de trabalho do Subversion são considerados detalhes de implementação realmente não planejados para consumo humano. Os desenvolvedores são encorajados a usar as APIs públicas do Subversion, ou as ferramentas oferecidas

pelo Subversion, para acessar e manipular os dados da cópia de trabalho, em vez de ler ou modificar esses arquivos diretamente. Os formatos de arquivo empregados pela biblioteca de cópia local para seus dados administrativos mudam de tempos em tempos—um fato em que as APIs públicas fazem um bom trabalho ao esconder do usuário comum. Nesta seção, vamos expor alguns destes detalhes de implementação para satisfazer sua imensa curiosidade.

Os Arquivos de Entrada

Talvez o mais importante arquivo no diretório `.svn` seja o arquivo `entries`. Ele contém a maior parte das informações administrativas sobre os ítens versionados em um diretório da cópia de trabalho. É este único arquivo que registra as URLs do repositório, revisão inalterada, *checksum* dos arquivos, textos inalterados e *timestamp* das propriedades, informações de agendamento e conflitos, últimas informações conhecida de submissão (autor, revisão, *timestamp*), histórico da cópia local—praticamente tudo que um cliente Subversion tem interesse em saber sobre um recurso versionado (ou a ser versionado)!

Pessoas familiares com os diretórios administrativos do CVS terão reconhecido neste ponto que o arquivo `.svn/entries` do Subversion serve o propósito, entre outras coisas, dos arquivos `CVS/Entries`, `CVS/Root` e `CVS/Repository` do CSV combinados.

O formato do arquivo `.snv/entries` tem mudado ao longo do tempo. Originalmente um arquivo XML, ele agora usa um formato de arquivo customizado—apesar de ainda legível a humanos. Enquanto o XML foi uma boa escolha no início para os desenvolvedores do Subversion que foram freqüentemente depurando o conteúdo do arquivo (e o comportamento do Subversion em função dele), a necessidade de uma depuração fácil no desenvolvimento foi se reduzindo com a maturidade do Subversion, e vem sendo substituída pela necessidade de desempenho do usuário. Fique ciente que a biblioteca de cópia de trabalho do Subversion atualiza automaticamente as cópias de trabalho de um formato para outro—ela lê os formatos antigos, e escreve os novos—o que salva você do problema de verificar uma nova cópia de trabalho, mas pode complicar em situações onde diferentes versões do Subversion podem estar tentando usar a mesma cópia de trabalho.

Cópias Inalteradas e Propriedade de Arquivos

Como mencionado antes, o diretório `.svn` também mantém as versões "text-base" intactas dos arquivos. Eles podem ser encontrados em `.svn/text-base`. Os benefícios destas cópias inalteradas são muitas—checagens de modificações locais e avisos de diferenças livres de rede, reversão de arquivos modificados ou perdidos livre de rede, maior eficiência na transmissão das mudanças para o servidor—mas tem o custo de possuir cada arquivo versionado armazenado duas vezes no disco. Nos dias de hoje, isso parece ser uma penalidade inofensiva para a maioria dos arquivos. Entretanto, a situação fica feia quando o tamanho de seus arquivos versionados aumenta. Uma atenção vem sendo dada para tornar opcional a presença do "text-base". Ironicamente, é quando o tamanho dos seus arquivos versionados fica maior que a existência do "text-base" se torna mais crucial—quem quer transmitir um arquivo enorme através de uma rede apenas porque precisa submeter uma pequena mudança para ele?

Com propósito similar aos arquivos "text-base" estão os arquivos de propriedade e suas cópias "prop-base" intactas, localizadas em `.svn/props` e `.svn/prop-base` respectivamente. Já que diretórios podem ter propriedades, também existem os arquivos `.svn/dir-props` e `.svn/dir-prop-base`.

Usando as APIs

Desenvolver aplicações junto com as APIs de biblioteca do Subversion é bastante simples. O Subversion é essencialmente um conjunto de bibliotecas C, com arquivos de cabeçalho (.h) que ficam no diretório `subversion/include` da árvore de fontes. Estes cabeçalhos são copiados para locais de seu sistema (por exemplo, `/usr/local/include`) quando você constrói e instala o Subversion a partir dos fontes. Estes cabeçalhos representam o conjunto das funções e tipos que são acessíveis pelos usuários das bibliotecas do Subversion. A comunidade de desenvolvedores do Subversion

é meticulosa em garantir que a API pública esteja bem documentada—refere-se diretamente aos arquivos de cabeçalho para esta documentação.

Ao examinar os arquivos de cabeçalho públicos, a primeira coisa que você pode perceber é que os tipos de dado e funções do Subversion possuem um espaço de nomes protegidos. Isto é, cada nome de símbolo público do Subversion inicia com `svn_`, seguido por um código curto para a biblioteca na qual o símbolo está definido (como `wc`, `client`, `fs`, etc.), seguido por um único sublinhado (_) e então o restante do nome do símbolo. As funções semi-públicas (usadas entre os arquivos fonte de uma dada biblioteca mas não por código fora desta biblioteca, e encontrada dentro de seus próprios diretórios) diferem deste esquema de nomeação em que, em vez de um único sublinhado depois do código da biblioteca, elas usam um sublinhado duplo (__). As funções que são privadas a um dado arquivo fonte não possuem prefixação especial, e são declaradas como `static`. Evidentemente, um compilador não está interessado nestas convenções de nomeação, mas elas ajudam esclarecer o escopo de uma dada função ou tipo de dado.

Uma outra boa fonte de informações sobre programação com as APIs do Subversion são as diretrizes de “hacking” do projeto, o qual pode ser encontrado em http://subversion.tigris.org/hacking.html. Este documento contém informações úteis que, embora destinadas a desenvolvedores e aos próprios desenvolvedores do Subversion, é igualmente aplicável a pessoas desenvolvendo com o Subversion como um conjunto de bibliotecas de terceiros. [2]

A Biblioteca Apache Portable Runtime

Juntamente com os tipos de dado do Subversion, você verá muitas referências a tipos de dado que iniciam com `apr_`—símbolos da biblioteca Apache Portable Runtime (APR). APR é uma biblioteca de portabilidade da Apache, originalmente esculpida fora do código de seu servidor como uma tentativa em separar as partes específicas de SO das porções independentes de SO. O resultado foi uma biblioteca que oferece uma API genérica para executar operações que diferem levemente—ou agressivamente—de SO para SO. Enquanto o Servidor HTTP da Apache foi obviamente o primeiro usuário da biblioteca APR, os desenvolvedores do Subversion imediatamente reconheceram o valor de usar APR também. Isto significa que existe praticamente nenhum código específico de SO no Subversion. Além disso, significa que o cliente Subversion compila e executa em qualquer lugar em que o Servidor HTTP da Apache compila e executa também. Atualmente esta lista inclui todos os sabores de Unix, Win32, BeOS, OS/2, e Mac OS X.

Além de oferecer implementações consistentes de chamadas de sistemas que diferem entre os sistemas operacionais, [3] a APR dá ao Subversion acesso imediato a vários tipos de dado personalizados, como matrizes dinâmicas e tabelas hash. O Subversion usa estes tipos extensivamente. Mas talvez o mais difundido tipo de dado da APR, encontrado em quase todo protótipo da API do Subversion, seja o apr_pool_t—o recipiente de memória da APR. O Subversion usa recipientes internamente para todas as suas necessidades de alocação de memória (a menos que uma biblioteca externa requeira um mecanismo de gerenciamento de memória diferente para que dados passem através de sua API), [4] e enquanto uma pessoa codifica com as APIs do Subversion não é necessário fazer o mesmo, eles *são* requeridos para fornecer recipientes para as funções da API que precisam deles. Isto significa que usuários da API do Subversion devem também vincular à APR, devem chamar `apr_initialize()` para inicializar o subsistema da APR, e então devem criar e gerenciar os recipientes para usar com as chamadas da API do Subversion, normalmente pelo uso de `svn_pool_create()`, `svn_pool_clear()`, e `svn_pool_destroy()`.

[2]Afinal, o Subversion usa as APIs do Subversion, também.
[3]O Subversion usa chamadas de sistema e tipos de dado em ANSI sempre que possível.
[4]Neon e Berkeley DB são exemplos de tais bibliotecas.

Programando com Recipientes de Memória

Quase todo desenvolvedor que usou a linguagem de programação C teve em algum ponto suspirou fundo ao enfrentar a assustadora tarefa de gerenciar o uso de memória. Alocar memória suficiente para o uso, manter controle dessas alocações, liberar a memória quando você não precisa mais dela—estas tarefas podem ser bastante complexas. E certamente, falhar ao fazer essas coisas adequadamente pode resultar em um programa que trava sozinho, ou pior ainda, trava todo o computador.

Em linguagens de alto nível, por outro lado, deixam o trabalho de gerenciamento de memória completamente distante do desenvolvedor. [5] Linguagens como Java e Python usam um *coletor de lixo*, alocando memória para os objetos quando necessário, e automaticamente liberando esta memória quando o objeto não está mais em uso.

A APR fornece uma abordagem de meio-termo chamada gerenciamento de memória baseada em recipientes. Isto permite ao desenvolvedor controlar o uso de memória em uma resolução mais baixa—por pedaços (ou "recipientes") de memória, em vez de por objeto alocado. Ao invés de usar `malloc()` e similares para alocar memória suficiente para um dado objeto, você pede que a APR aloque a memória de um recipiente de memória. Quando você estiver finalizado o uso dos objetos criados naquele recipiente, você destrói todo o recipiente, efetivamente desalocando a memória consumida por *todos* os objetos que você alocou nele. Dessa forma, em vez de manter o controle individual de objetos que precisam ser desalocados, seu programa simplesmente considera o tempo de vida total desses objetos, e aloca os objetos em um recipiente cujo tempo de vida (o tempo entre a criação do recipiente e sua exclusão) coincide a necessidade do objeto.

Requisitos de URL e Caminho

Com operação remota de controle de versão como o ponto de toda a existência do Subversion, faz sentido que alguma atenção seja dada ao suporte de internacionalização (i18n). Afinal, enquanto o "remoto" possa significar "em todo o escritório", poderia perfeitamente dizer "em todo o globo". Para facilitar isto, todas as interfaces públicas do Subversion que aceitam argumentos de caminho esperam que esses caminhos sejam canonicalizados e codificados em UTF-8. Isto significa, por exemplo, que qualquer novo cliente binário que realiza a interface libsvn_client precisa primeiro converter os caminhos a partir da codificação específica da localidade para a UTF-8 antes de passar esses caminhos para as bibliotecas do Subversion, e então reconverter qualquer dos caminhos de saída resultantes do retorno do Subversion para a codificação da localidade antes de usar esses caminhos para propósitos fora do Subversion. Felizmente, o Subversion oferece um conjunto de funções (veja `subversion/include/svn_utf.h`) que podem ser usadas por qualquer programa para fazer esses conversões.

Além disso, as APIs do Subversion requerem que todos os parâmetros da URL sejam devidamente codificados em URI. Assim, em vez de passar `file:///home/username/My File.txt` como a URL de um arquivo nomeado `My File.txt`, você precisa passar `file:///home/username/My%20File.txt`. Novamente, o Subversion fornece funções auxiliares que sua aplicação pode usar—`svn_path_uri_encode()` e `svn_path_uri_decode()`, para codificação e decodificação de URI, respectivamente.

Usando Outras Linguagens além de C e C++

Se você está interessado em usar as bibliotecas do Subversion em conjunção com alguma outra coisa do que um programa C—digo um script Python ou Perl—o Subversion possui algum suporte para isto por meio do *Simplified Wrapper and Interface Generator* (SWIG). Os vínculos do SWIG com o Subversion estão localizados em `subversion/bindings/swig`. Eles estão ainda amadurecendo, mas já são usáveis. Estes vínculos permitem você chamar as funções da API do Subversion indiretamente, usando invólucros que traduzem os tipos de dado nativos de sua linguagem de *scripting* para os tipos de dado necessários das bibliotecas C do Subversion.

[5] Ou pelo menos torná-lo algo que você somente diverte-se ao fazer uma otimização extremamente rígida do programa.

Esforços significantes vêm sendo realizados na criação de vínculos funcionais gerados por SWIG para Python, Perl e Ruby. De certa forma, o trabalho realizado preparando os arquivos de interface SWIG para estas linguagens é reutilizável em esforços para produzir vínculos para outras linguagens suportadas por SWIG (as quais incluem versões de C#, Guile, Java, MzScheme, OCaml, PHP, e Tcl, entre outras). No entanto, alguma programação extra é necessária para compensar as APIs complexas, assim o SWIG precisa de alguma ajuda na tradução entre linguagens. Para mais informações sobre o SWIG, veja o site do projeto em http://www.swig.org/.

O Subversion também possui vínculos de linguagem para Java. Os vínculos JavaJL (localizados em `subversion/bindings/java` na árvore de fontes do Subversion) não são baseados no SWIG, porém são uma mistura de javah e JNI codificada na unha. JavaHL abrange a maior parte das APIs do Subversion no lado do cliente, e é especificamente orientada aos implementadores de clientes Subversion baseado em Java e integrações em IDE.

Os vínculos de linguagem do Subversion tendem a necessitar do nível de atenção do desenvolvedor dada aos módulos principais do Subversion, mas podem geralmente serem confiáveis como prontos para produção. Um número de scripts e aplicações, clientes Subversion alternativos com GUI e outras ferramentas de terceiros estão atualmente usando com sucesso os vínculos de linguagem do Subversion para realizar suas integrações com o Subversion.

Cabe notar aqui que existem outras opções para interfacear com o Subversion usando outras linguagens: vínculos alternativos para o Subversion que não são fornecidos por toda a comunidade de desenvolvimento do Subversion. Você pode encontrar links para estes vínculos alternativos na página de links do projeto Subversion (em http://subversion.tigris.org/links.html), mas existe uma dupla popular que sentimos serem especialmente notáveis. Primeiro, os vínculos PySVN de Barry Scott (http://pysvn.tigris.org/) são uma opção popular para vinculação com Python. PySVN ostenta uma interface mais "Pythônica" do que a das APIs baseadas na da C e oferecida pelos vínculos Python do próprio Subversion. Para pessoas procurando por uma implementação puramente em Java do Subversion, verifiquem o SVNKit (http://svnkit.com/), que é um Subversion reescrito totalmente em Java. Contudo, você deve ter muito cuidado aqui—porque o SVNKit não utiliza as bibliotecas base do Subversion, seu comportamento não possui garantias de coincidir com o do próprio Subversion.

Exemplos de Código

Exemplo 8.1, "Usando a Camada do Repositório" contém um segmento de código (escrito em C) que ilustra alguns dos conceitos que estamos discutindo. Ele usa ambas as interfaces de repositório e sistema de arquivo (como pode ser determinado pelos prefixos `svn_repos_` e `svn_fs_` dos nomes de função, respectivamente) para criar uma nova revisão na qual um diretório é adicionado. Você pode ver o uso de um recipiente APR, o qual é passado para propósitos de alocação de memória. Além disso, o código revela um fato um tanto obscuro sobre tratamento de erros do Subversion—todos os erros do Subversion devem ser explicitamente tratados para evitar vazamento de memória (e em alguns casos, falha da aplicação).

Exemplo 8.1. Usando a Camada do Repositório

```
  /* Abre o repositório localizado em REPOS_PATH.
   */
  INT_ERR(svn_repos_open(&repos, repos_path, pool));

  /* Obtém um ponteiro para o objeto de sistema de arquivo armazenado em REPOS.
   */
  fs = svn_repos_fs(repos);

  /* Pede ao sistema de arquivo para nos retornar a mais jovem revisão que
   * existe atualmente.
   */
  INT_ERR(svn_fs_youngest_rev(&youngest_rev, fs, pool));

  /* Inicia uma nova transação que tem por base a YOUNGEST_REV.  Nós estamos
   * menos prováveis de ter nossa submissão rejeitada como conflitante se
   * sempre tentarmos fazer nossas mudanças novamente em uma cópia da última
   * imagem da árvore do sistema de arquivo.
   */
  INT_ERR(svn_fs_begin_txn(&txn, fs, youngest_rev, pool));

  /* Agora que temos iniciada uma nova transação Subversion, recupera um objeto
   * raíz que representa esta transação.
   */
  INT_ERR(svn_fs_txn_root(&txn_root, txn, pool));

  /* Cria nosso novo diretório sob a transação raíz, para o caminho
   * NEW_DIRECTORY.
   */
  INT_ERR(svn_fs_make_dir(txn_root, new_directory, pool));

  /* Submete a transação, criando uma nova revisão do sistema de arquivo
   * a qual inclui o caminho de nosso diretório adicionado.
   */
  err = svn_repos_fs_commit_txn(&conflict_str, repos,
                                &youngest_rev, txn, pool);
  if (! err)
    {
      /* Sem erro?  Excelente!  Imprime um breve relatório de nosso sucesso.
       */
      printf("O diretório '%s' foi adicionado com sucesso na nova revisão "
             "'%ld'.\n", new_directory, youngest_rev);
    }
  else if (err->apr_err == SVN_ERR_FS_CONFLICT)
    {
      /* Oh não.  Nossa submissão falhou como resultado de um conflito
       * (alguém parece ter feito mudanças na mesma área do sistema de
       * arquivo que nós tentamos modificar).  Imprime uma mensagem de
       * erro.
       */
      printf("Um conflito ocorreu no caminho '%s' na tentativa de "
             "adicionar o diretório '%s' no repositório em '%s'.\n",
             conflict_str, new_directory, repos_path);
    }
  else
    {
      /* Algum outro erro ocorreu.  Imprime uma mensagem de erro.
       */
      printf("Um erro ocorreu na tentativa de adicionar o diretório '%s' "
             "no repositório em '%s'.\n",
             new_directory, repos_path);
    }

  INT_ERR(err);
}
```

Note que em Exemplo 8.1, “Usando a Camada do Repositório”, o código poderia ter apenas tão facilmente submetido a transação usando `svn_fs_commit_txn()`. Mas a API do sistema de arquivo sabe nada sobre o mecanismo de gancho da biblioteca do repositório. Se você quer que seu repositório Subversion realize automaticamente algum conjunto de tarefas não-Subversion toda vez que você submeter uma transação (como, por exemplo, enviar um email que descreve todas as mudanças feitas nesta transação para sua lista de discussão de desenvolvedores), você precisa usar a versão desta função embrulhada em libsvn_repos, a qual adiciona a funcionalidade de disparo de gancho—neste caso, `svn_repos_fs_commit_txn()`. (Para mais informações em relação aos ganchos de repositório Subversion, veja “Implementing Repository Hooks”.)

Agora vamos trocar as linguagens. Exemplo 8.2, “Using the Repository Layer with Python” é um exemplo de programa que usa os vínculos Python SWIG do Subversion para recursivamente rastrear a mais jovem revisão do repositório, e imprimir os vários caminhos descobertos durante o rastreamento.

Exemplo 8.2. Using the Repository Layer with Python

```
#!/usr/bin/python

"""Crawl a repository, printing versioned object path names."""

import sys
import os.path
import svn.fs, svn.core, svn.repos

def crawl_filesystem_dir(root, directory):
    """Recursively crawl DIRECTORY under ROOT in the filesystem, and return
    a list of all the paths at or below DIRECTORY."""

    # Print the name of this path.
    print directory + "/"

    # Get the directory entries for DIRECTORY.
    entries = svn.fs.svn_fs_dir_entries(root, directory)

    # Loop over the entries.
    names = entries.keys()
    for name in names:
        # Calculate the entry's full path.
        full_path = directory + '/' + name

        # If the entry is a directory, recurse.  The recursion will return
        # a list with the entry and all its children, which we will add to
        # our running list of paths.
        if svn.fs.svn_fs_is_dir(root, full_path):
            crawl_filesystem_dir(root, full_path)
        else:
            # Else it's a file, so print its path here.
            print full_path

def crawl_youngest(repos_path):
    """Open the repository at REPOS_PATH, and recursively crawl its
    youngest revision."""

    # Open the repository at REPOS_PATH, and get a reference to its
    # versioning filesystem.
    repos_obj = svn.repos.svn_repos_open(repos_path)
    fs_obj = svn.repos.svn_repos_fs(repos_obj)

    # Query the current youngest revision.
    youngest_rev = svn.fs.svn_fs_youngest_rev(fs_obj)

    # Open a root object representing the youngest (HEAD) revision.
    root_obj = svn.fs.svn_fs_revision_root(fs_obj, youngest_rev)

    # Do the recursive crawl.
    crawl_filesystem_dir(root_obj, "")

if __name__ == "__main__":
    # Check for sane usage.
    if len(sys.argv) != 2:
        sys.stderr.write("Usage: %s REPOS_PATH\n"
                         % (os.path.basename(sys.argv[0])))
        sys.exit(1)

    # Canonicalize the repository path.
    repos_path = svn.core.svn_path_canonicalize(sys.argv[1])

    # Do the real work.
    crawl_youngest(repos_path)
```

This same program in C would need to deal with APR's memory pool system. But Python handles memory usage automatically, and Subversion's Python bindings adhere to that convention. In C, you'd be working with custom datatypes (such as those provided by the APR library) for representing the hash of entries and the list of paths, but Python has hashes (called “dictionaries”) and lists as built-in datatypes, and provides a rich collection of functions for operating on those types. So SWIG (with the help of some customizations in Subversion's language bindings layer) takes care of mapping those custom datatypes into the native datatypes of the target language. This provides a more intuitive interface for users of that language.

The Subversion Python bindings can be used for working copy operations, too. In the previous section of this chapter, we mentioned the `libsvn_client` interface, and how it exists for the sole purpose of simplifying the process of writing a Subversion client. Exemplo 8.3, “A Python Status Crawler” is a brief example of how that library can be accessed via the SWIG Python bindings to recreate a scaled-down version of the **svn status** command.

Exemplo 8.3. A Python Status Crawler

```
                 svn.wc.svn_wc_status_modified    : 'M',
                 svn.wc.svn_wc_status_merged      : 'G',
                 svn.wc.svn_wc_status_conflicted  : 'C',
                 svn.wc.svn_wc_status_obstructed  : '~',
                 svn.wc.svn_wc_status_ignored     : 'I',
                 svn.wc.svn_wc_status_external    : 'X',
                 svn.wc.svn_wc_status_unversioned : '?',
               }
    return code_map.get(status, '?')

def do_status(wc_path, verbose):
    # Calculate the length of the input working copy path.
    wc_path_len = len(wc_path)

    # Build a client context baton.
    ctx = svn.client.svn_client_ctx_t()

    def _status_callback(path, status, root_path_len=wc_path_len):
        """A callback function for svn_client_status."""

        # Print the path, minus the bit that overlaps with the root of
        # the status crawl
        text_status = generate_status_code(status.text_status)
        prop_status = generate_status_code(status.prop_status)
        print '%s%s  %s' % (text_status, prop_status, path[wc_path_len + 1:])

    # Do the status crawl, using _status_callback() as our callback function.
    svn.client.svn_client_status(wc_path, None, _status_callback,
                                 1, verbose, 0, 0, ctx)

def usage_and_exit(errorcode):
    """Print usage message, and exit with ERRORCODE."""
    stream = errorcode and sys.stderr or sys.stdout
    stream.write("""Usage: %s OPTIONS WC-PATH
Options:
  --help, -h    : Show this usage message
  --verbose, -v : Show all statuses, even uninteresting ones
""" % (os.path.basename(sys.argv[0])))
    sys.exit(errorcode)

if __name__ == '__main__':
    # Parse command-line options.
    try:
        opts, args = getopt.getopt(sys.argv[1:], "hv", ["help", "verbose"])
    except getopt.GetoptError:
        usage_and_exit(1)
    verbose = 0
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage_and_exit(0)
        if opt in ("-v", "--verbose"):
            verbose = 1
    if len(args) != 1:
        usage_and_exit(2)

    # Canonicalize the repository path.
    wc_path = svn.core.svn_path_canonicalize(args[0])

    # Do the real work.
    try:
        do_status(wc_path, verbose)
    except svn.core.SubversionException, e:
        sys.stderr.write("Error (%d): %s\n" % (e[1], e[0]))
        sys.exit(1)
```

As was the case in Exemplo 8.2, “Using the Repository Layer with Python”, this program is pool-free and uses, for the most part, normal Python data types. The call to `svn_client_ctx_t()` is deceiving because the public Subversion API has no such function—this just happens to be a case where SWIG's automatic language generation bleeds through a little bit (the function is a sort of factory function for Python's version of the corresponding complex C structure). Also note that the path passed to this program (like the last one) gets run through `svn_path_canonicalize()`, because to *not* do so runs the risk of triggering the underlying Subversion C library's assertions about such things, which translate into rather immediate and unceremonious program abortion.

Capítulo 9. Referência Completa do Subversion

Este capítulo tem a intenção de ser uma referência completa para o uso do Subversion. Ele inclui o comando (**svn**) e todos os seus subcomandos, assim como programas de administração de repositório (**svnadmin** e **svnlook**) e seus respectivos sub-comandos.

O Cliente de Linha de Comando do Subversion: svn

Para usar o cliente de linha de comando, você digita **svn**, o subcomando que você quer usar [1], e quaisquer opções ou argumentos de destino sobre os quais você quer realizar a operação—não há uma ordem específica na qual o subcomando e as opções devam aparecer. Por exemplo, todos os comandos a seguir são formas válidas de se usar um **svn status**:

```
$ svn -v status
$ svn status -v
$ svn status -v meuarquivo
```

Você pode encontrar muitos exemplos de como usar a maioria dos comandos do cliente Subversion Capítulo 2, *Uso Básico* e dos comandos para gerenciamento de propriedades em “Propriedades”.

Opções do svn

Ainda que o Subversion tenha diferentes opções para seus subcomandos, todas as opções são globais —isto é, garante-se que cada opção signifique a mesma coisa independentemente do subcomando que você use com ela. Por exemplo, `--verbose` (`-v`) sempre significa “saída verbosa”, qualquer que seja o subcomando que você utilizar com ela.

`--auto-props`
: Habilita auto-props, sobrescrevendo a diretiva `enable-auto-props` no arquivo `config`.

`--change` (`-c`) *`ARG`*
: Usado como uma forma de se referir a uma “mudança” (leia-se uma revisão), esta opção é sintaticamente equivalente a “-r ARG-1:ARG”.

`--config-dir` *`DIR`*
: Diz para o Subversion ler a informação de configuração a partir do diretório especificado ao invés de seu local padrão (`.subversion` no diretório home do usuário).

`--diff-cmd` *`CMD`*
: Especifica um programa externo a ser usado para exibir diferenças entre arquivos. Quando **svn diff** é invocado sem esta opção, ele usa o mecanismo de diff interno do Subversion, que exibe diffs unificados por padrão. Se você quiser usar um programa diff externo, use `--diff-cmd`. Você pode passar opções para o programa diff com a opção `--extensions` (mais detalhes sobre isso mais adiante nesta seção).

`--diff3-cmd` *`CMD`*
: Especifica um programa externo a ser usado para mesclar arquivos.

`--dry-run`
: Faz menção de todos os passos de execução de um comando, mas sem efetuar qualquer alteração —tanto no disco quanto no repositório.

[1] Sim, sim, você não precisa de um subcomando para usar a opção `--version`, mas vamos falar disso num instante.

`--editor-cmd` *CMD*
: Especifica um programa externo a ser usado para editar mensagens de log ou valores de propriedades. Consulte a seção `editor-cmd` em “Configuração” para ver as formas de especificar um editor padrão.

`--encoding` *ENC*
: Informa ao Subversion que sua mensagem de submissão está codificada com o charset dado. O padrão é o locale nativo de seu sistema operacional, e você deve especificar a codificação se sua mensagem de commit estiver em alguma codificação diferente.

`--extensions` (`-x`) *ARGS*
: Especifica um argumento ou argumentos que o Subversion deve passar para um comando diff externo. Esta opção é valida apenas quando usada com os comando **svn diff** ou **svn merge**, com a opção `--diff-cmd`. Se você quiser passar argumentos múltiplos, você deve delimitá-los todos entre aspas duplas (por exemplo, **svn diff --diff-cmd /usr/bin/diff -x "-b -E"**).

`--file` (`-F`) *FILENAME*
: Usa o conteúdo do arquivo especificado para o subcomando em questão, sendo que subcomandos diferentes fazem coisas diferentes com este conteúdo. Por exemplo, o **svn commit** usa o conteúdo como uma mensagem de log do commit, ao passo que o **svn propset** o utiliza como um valor de propriedade.

`--force`
: Força a execução de um comando ou operação específicos. Há algumas operações em que o Subversion irá impedi-lo de prosseguir, em sua utilização normal, mas você pode passar a opção force para dizer ao Subversion “Eu sei o que estou fazendo, bem como as possíveis consequências disto, então deixe-me fazê-lo”. Esta opção seria o equivalente a fazer você mesmo um reparo elétrico com a energia ligada—se você não souber o que está fazendo, é provável que tome um choque desagradável.

`--force-log`
: Força que um parâmetro suspeito passado para às opções `--message` (`-m`) ou `--file` (`-F`) seja aceito como válido. Por padrão, o Subversion irá produzir um erro se os parâmetros destas opções parecerem com argumentos de destino do subcomando. Por exemplo, se você passar um caminho de um arquivo versionado para a opção `--file` (`-F`), o Subversion irá assumir que você cometeu um engano, que caminho informado pretendia ser o objeto alvo da operação, e que você simplesmente esqueceu de informar algum outro arquivo—não-versionado—contendo sua mensagem de log. Para confirmar sua intenção e sobrescrever esse tipo de erro, passe a opção `--force-log` para os subcomandos que aceitam mensagens de log.

`--help` (`-h` or `-?`)
: Se usado com um ou mais subcomandos, mostra o texto de ajuda preexistente para cada subcomando. Se usado sozinho, exibe o texto de ajuda geral para o cliente Subversion.

`--ignore-ancestry`
: Diz para o Subversion ignorar ancestrais (diretórios acima) ao determinar diferenças (baseia-se apenas em conteúdos dos caminhos).

`--ignore-externals`
: Diz para o Subversion ignorar definições externas e cópias de trabalho externas gerenciadas por elas.

`--incremental`
: Exibe saída em um formado adequado para concatenação.

`--limit` *NUM*
: Exibe apenas as primeiras *NUM* mensagens de log.

`--message` (`-m`) *MESSAGE*
: Indica que você irá especificar seja uma mensagem de log ou seja um comentário de uma trava na linha de comando, depois desta opção. Por exemplo:

```
$ svn commit -m "Eles não fazem isso no domingo."
```

`--new` *`ARG`*
: Utiliza *`ARG`* como objeto alvo novo (para uso com **svn diff**).

`--no-auth-cache`
: Evita cache de informação de autenticação (p.ex. nome de usuário e senha) nos diretórios administrativos do Subversion.

`--no-auto-props`
: Desabilita auto-props, sobrescrevendo a diretiva `enable-auto-props` no arquivo `config`.

`--no-diff-added`
: Evita que o Subversion exiba diferenças para arquivos adicionados. O comportamento padrão ao adicionar um arquivo é que o **svn diff** exiba as mesmas diferenças que você veria se tivesse adicionado todo o conteúdo a um arquivo (vazio) existente.

`--no-diff-deleted`
: Evita que o Subversion exiba diferenças para arquivos excluídos. O comportamento padrão ao remover um arquivo é que o **svn diff** exiba as mesmas diferenças que você veria se tivesse mantido o arquivo mas removido todo o seu conteúdo.

`--no-ignore`
: Mostra arquivos na listagem de status que normalmente seriam omitidos por corresponderem a um padrão na opção de configuração `global-ignores` ou na propriedade `svn:ignore`. Veja “Configuração” e “Ignorando Itens Não-Versionados” para mais informações.

`--no-unlock`
: Não destrava arquivos automaticamente (o comportamento padrão é destravar todos os arquivos listados como parte de um commit). Veja “Travamento” para mais informações.

`--non-interactive`
: No caso de uma falha de autenticação, ou credenciais insuficientes, evita a outra solicitação de credenciais (p.ex. nome de usuário ou senha). É útil se você estiver executando o Subversion dentro de um script automatizado e for mais adequado ter uma falha de autenticação do Subversion do que um prompt solicitando mais informação.

`--non-recursive` (`-N`)
: Evita a recursão de um subcomando dentro de subdiretórios. Muitos subcomandos fazem recursão por padrão, mas alguns subcomandos—usualmente aqueles com potencial de remover ou desfazer suas alterações locais—não.

`--notice-ancestry`
: Leva os ancestrais (diretórios acima) em consideração ao determinar diferenças.

`--old` *`ARG`*
: Utiliza *`ARG`* como objeto alvo antigo (para uso com **svn diff**).

`--password` *`PASS`*
: Indica que você está informando sua senha para autenticação na linha de comando—do contrário, se for preciso, o Subversion irá solicitá-la interativamente.

`--quiet` (`-q`)
: Solicita que o cliente exiba apenas as informações mais essenciais ao executar uma dada operação.

`--recursive` (`-R`)
: Faz um subcomando executar recursivamente dentro dos subdiretórios. A maioria dos subcomandos já executam recursivamente por padrão.

`--relocate` *FROM TO [PATH...]*
Usado com o subcomando **svn switch**, troca o local do repositório ao qual sua cópia de trabalho se refere. Isto é útil se o local de seu repositório muda e você tem uma cópia de trabalho existente que gostaria de continuar a usar. Consulte **svn switch** para conferir um exemplo.

`--revision` (`-r`) *REV*
Indica que você estará informando uma revisão (ou intervalo de revisões) para uma dada operação. Você pode informar números de revisão, termos de revisão ou datas (entre chaves) como argumentos para a opção de revisão. Se você quiser fornecer um intervalo de revisões, você pode informar duas revisões separadas por dois-pontos. Por exemplo:

```
$ svn log -r 1729
$ svn log -r 1729:HEAD
$ svn log -r 1729:1744
$ svn log -r {2001-12-04}:{2002-02-17}
$ svn log -r 1729:{2002-02-17}
```

Veja “Termos de Revisão” para informações.

`--revprop`
Opera em uma propriedade de revisão ao invés de uma propriedade específica de um arquivo ou diretório. Esta opção requer que você também informe uma revisão com a opção `--revision` (`-r`).

`--show-updates` (`-u`)
Faz com que o cliente exiba informação sobre quais arquivos em sua cópia de trabalho estão desatualizados. Isto não atualiza nenhum de seus arquivos—apenas mostra a você quais arquivos serão atualizados se você executar um **svn update**.

`--stop-on-copy`
Faz com que um subcomando do Subversion que está varrendo o histórico de um recurso versionado pare de vasculhar essas informações históricas quando uma cópia—isto é, um local no histórico em que o recurso foi copiado de outro local no repositório—seja encontrada.

`--strict`
Faz com que o Subversion use semântica estrita, um conceito que é um pouco vago a não ser quando aplicado a subcomandos específicos (especificamente ao **svn propget**).

`--targets` *FILENAME*
Diz para o Subversion obter a lista de arquivos em que você quer executar uma operação a partir de um nome de arquivo informado ao invés de ter de listar todos os arquivos na linha de comando.

`--username` *NAME*
Indica que você está informando seu nome de usuário para autenticação na linha de comando—de outra forma, se for necessário, o Subversion irá solicitá-lo a você interativamente.

`--verbose` (`-v`)
Solicita que o cliente exibe o máximo de informação que puder ao executar qualquer subcomando. Isto pode fazer com que o Subversion exiba campos adicionais, informações detalhadas sobre cada arquivo, ou informações extras sobre suas ações em execução.

`--version`
Exibe informação da versão do cliente. Esta informação não apenas inclui o número de versão do cliente, mas também uma listagem de todos os módulos de acesso ao repositório que o cliente pode usar para acessar um repositório Subversion. Com `--quiet` (`-q`) ele exibe apenas o número de versão de forma compacta.

`--xml`
Exibe a saída em um formato XML.

Subcomandos svn

Aqui temos vários subcomandos:

Nome

svn add — Adiciona arquivos, diretórios ou links simbólicos.

Sinopse

```
svn add PATH...
```

Descrição

Agenda arquivos, diretórios ou links simbólicos em sua cópia de trabalho para adição no repositório. Eles serão carregados e adicionados efetivamente ao repositório em sua próxima submissão. Se você adicionar alguma coisa e mudar de idéia antes de realizar o commit, você pode cancelar o agendamento usando **svn revert**.

Nomes Alternativos

Nenhum

Altera

Cópia de trabalho

Acessa o Repositório

Não

Opções

```
--targets FILENAME
--non-recursive (-N)
--quiet (-q)
--config-dir DIR
--no-ignore
--auto-props
--no-auto-props
--force
```

Exemplos

Para adicionar um arquivo para sua cópia de trabalho:

```
$ svn add foo.c
A         foo.c
```

Ao adicionar um diretório, o comportamento padrão do **svn add** é ser recursivo:

```
$ svn add testdir
A         testdir
A         testdir/a
A         testdir/b
A         testdir/c
A         testdir/d
```

Você também pode adicionar um diretório sem adicionar seu conteúdo:

```
$ svn add --non-recursive otherdir
A         otherdir
```

Normalmente, o comando **svn add *** vai desconsiderar quaisquer diretórios que já estiverem sob controle de versão. Algumas vezes, no entanto, você pode querer adicionar quaisquer objetos não-versionados em sua cópia de trabalho, incluindo os que estiverem escondidos mais profundamente na árvore de diretórios. Passar a opção `--force` faz com que o **svn add** aja recursivamente também nos diretórios versionados:

```
$ svn add * --force
A         foo.c
A         somedir/bar.c
A         otherdir/docs/baz.doc
…
```

Nome

svn blame — Mostra informação de autor e revisão por linha para os arquivos ou URLs especificados.

Sinopse

```
svn blame TARGET[@REV]...
```

Descrição

Mostra informação de autor e revisão por linha para os arquivos ou URLs especificados. Cada linha de texto é prefixada com o nome do autor (nome de usuário) e o número de revisão da última alteração naquela linha.

Nomes Alternativos

praise, annotate, ann

Altera

Nada

Acessa o Repositório

Sim

Opções

```
--revision (-r) ARG
--verbose (-v)
--incremental
--xml
--extensions (-x) ARG
--force
--username ARG
--password ARG
--no-auth-cache
--non-interactive
--config-dir ARG
```

Exemplos

Se você quiser ver informação dos responsáveis pelos fontes para `readme.txt` em seu repositório:

```
$ svn blame http://svn.red-bean.com/repos/test/readme.txt
     3      sally This is a README file.
     5      harry You should read this.
```

Ainda que o **svn blame** informe que Harry modificou o arquivo readme.txt por último na revisão 5, você terá de verificar exatamente o que a revisão mudou para ter certeza de que Harry tenha mudado o *context* da linha—ele pode ter apenas ajustado espaços em branco, p.ex.

Nome

svn cat — Exibe o conteúdo de arquivos ou URLs especificadas.

Sinopse

```
svn cat TARGET[@REV]...
```

Descrição

Exibe o conteúdo de arquivos ou URLs especificadas. Para listar o conteúdo de diretórios, veja **svn list**.

Nomes Alternativos

Nenhum

Altera

Nada

Acessa o Repositório

Sim

Opções

```
--revision (-r) REV
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Exemplos

Se você quiser ver o arquivo readme.txt em seu repositório sem carregá-lo para sua cópia de trabalho (check out):

```
$ svn cat http://svn.red-bean.com/repos/test/readme.txt
This is a README file.
You should read this.
```

Se sua cópia de trabalho estiver desatualizada (ou se você tiver alterações locais) e você quiser ver a revisão `HEAD` de um arquivo em sua cópia de trabalho, o **svn cat** vai obter automaticamente a revisão `HEAD` quando você lhe passar um caminho:

```
$ cat foo.c
This file is in my local working copy
and has changes that I've made.

$ svn cat foo.c
Latest revision fresh from the repository!
```

Nome

svn checkout — Carrega uma cópia de trabalho a partir do repositório.

Sinopse

```
svn checkout URL[@REV]... [PATH]
```

Descrição

Carrega uma cópia de trabalho a partir do repositório. Se *PATH* for omitido, a base da URL será usada como destino. Se múltiplas URLs forem informadas, cada uma será carregada dentro de um subdiretório de *PATH*, com o nome do subdiretório sendo a base da URL.

Nomes Alternativos

co

Altera

Cria uma cópia de trabalho.

Acessa o Repositório

Sim

Opções

```
--revision (-r) REV
--quiet (-q)
--non-recursive (-N)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--ignore-externals
--config-dir DIR
```

Exemplos

Carrega uma cópia de trabalho dentro de subdiretório chamado `mine`:

```
$ svn checkout file:///tmp/repos/test mine
A  mine/a
A  mine/b
Checked out revision 2.
$ ls
mine
```

Carrega dois diretórios diferentes para duas cópias de trabalho separadas:

```
$ svn checkout file:///tmp/repos/test  file:///tmp/repos/quiz
A  test/a
A  test/b
Checked out revision 2.
A  quiz/l
```

```
A  quiz/m
Checked out revision 2.
$ ls
quiz  test
```

Carrega dois diretórios diferentes para duas cópias de trabalho separadas, mas põe ambas dentro de um diretório chamado `working-copies`:

```
$ svn checkout file:///tmp/repos/test  file:///tmp/repos/quiz working-copies
A  working-copies/test/a
A  working-copies/test/b
Checked out revision 2.
A  working-copies/quiz/l
A  working-copies/quiz/m
Checked out revision 2.
$ ls
working-copies
```

Se você interromper um checkout (ou houver qualquer coisa que o interrompa, como perda da conexão de rede, etc.), você pode reiniciá-lo tanto executando um comando checkout idêntico outra vez, como atualizando sua cópia de trabalho incompleta:

```
$ svn checkout file:///tmp/repos/test test
A  test/a
A  test/b
^C
svn: The operation was interrupted
svn: caught SIGINT

$ svn checkout file:///tmp/repos/test test
A  test/c
A  test/d
^C
svn: The operation was interrupted
svn: caught SIGINT

$ cd test
$ svn update
A  test/e
A  test/f
Updated to revision 3.
```

Nome

svn cleanup — Faz recursivamente uma limpeza na área de trabalho.

Sinopse

```
svn cleanup [PATH...]
```

Descrição

Faz recursivamente uma limpeza na área de trabalho, removendo travas da cópia de trabalho e concluindo operações não finalizadas. Sempre que você obtiver um erro de “cópia de trabalho travada”, execute este comando para remover travas perdidas e deixar sua cópia de trabalho no estado usável novamente.

Se, por algum motivo, um **svn update** falhar devido a um problema ao executar um programa diff externo (p.ex. devido a entrada do usuário ou falha de rede), passe a opção `--diff3-cmd` para permitir que a limpeza complete qualquer mesclagem com seu programa diff externo. Você também pode especificar um diretório de configuração com `--config-dir`, mas você deve precisar destas opções em raríssimas ocasições.

Nomes Alternativos

Nenhum

Altera

Cópia de trabalho

Acessa o Repositório

Não

Opções

```
--diff3-cmd CMD
--config-dir DIR
```

Exemplos

Bem, não há muito para esta parte de exemplo aqui pois o **svn cleanup** não gera nenhuma saída. Se você não informar um caminho (*PATH*), então “`.`” é usado.

```
$ svn cleanup

$ svn cleanup /path/to/working-copy
```

Nome

svn commit — Envia as alterações de sua cópia de trabalho para o repositório.

Sinopse

```
svn commit [PATH...]
```

Descrição

Envia as alterações de sua cópia de trabalho para o repositório. Se você não informar uma mensagem de log de registro usando seja a opção `--file` ou a opção `--message`, o **svn** vai executar seu editor de texto para que você escreva uma mensagem de commit. Veja a seção `editor-cmd` em “Configuração”.

svn commit vai enviar quaisquer marcadores de trava que encontrar e irá liberar as travas em todos os caminhos *PATHS* submetidos (recursivamente), a menos que `--no-unlock` seja passado.

Se você começar um commit e o Subversion executar seu editor para que você componha a mensagem de registro, você ainda pode abortar sem submeter suas alterações. Se você quiser cancelar a operação, apenas feche o editor sem salvar sua mensagem de log e o Subversion irá lhe perguntar se você quer cancelar a operação, prosseguir sem uma mensagem de log, ou editar a mensagem novamente.

Nomes Alternativos

ci (abreviação para “check in”; e não “co”, que é a abreviação para “checkout”)

Altera

Cópia de trabalho, repositório

Acessa o Repositório

Sim

Opções

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--no-unlock
--non-recursive (-N)
--targets FILENAME
--force-log
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--config-dir DIR
```

Exemplos

Submete uma modificação simples em um arquivo com a mensagem de log na linha de comando e um alvo implícito de seu diretório atual (“`.`”):

```
$ svn commit -m "added howto section."
```

```
Sending        a
Transmitting file data .
Committed revision 3.
```

Submete uma modificação no arquivo `foo.c` (explicitamente especificado na linha de comando) com a mensagem de log contida em um arquivo chamado `msg`:

```
$ svn commit -F msg foo.c
Sending        foo.c
Transmitting file data .
Committed revision 5.
```

Se você quiser usar um arquivo que está sob controle de versão para sua mensagem de log com `--file`, você precisa passar a opção `--force-log`:

```
$ svn commit --file file_under_vc.txt foo.c
svn: The log message file is under version control
svn: Log message file is a versioned file; use '--force-log' to override

$ svn commit --force-log --file file_under_vc.txt foo.c
Sending        foo.c
Transmitting file data .
Committed revision 6.
```

Para efetivar a remoção de um arquivo agendado para exclusão:

```
$ svn commit -m "removed file 'c'."
Deleting       c

Committed revision 7.
```

Nome

svn copy — Copia um arquivo ou diretório em uma cópia de trabalho ou no repositório.

Sinopse

```
svn copy SRC DST
```

Descrição

Copia um arquivo em uma cópia de trabalho ou no repositório. *SRC* e *DST* podem ser tanto um caminho em uma cópia de trabalho (WC) ou uma URL:

WC -> WC
: Copia e agenda um item para adição (com o histórico).

WC -> URL
: Imediatamente submete uma cópia de WC para a URL.

URL -> WC
: Obtém (*check out*) a URL para dentro da WC, agendando-a para adição.

URL -> URL
: Execução de uma cópia inteiramente do lado do servidor. Isto é frequentemente usado para criação de ramos (*branch*) e de rótulos (*tag*).

Você só pode copiar arquivos para dentro de um único repositório. O Subversion não dá suporte a cópias entre repositórios distintos.

Nomes Alternativos

cp

Altera

O repositório, se o destino for uma URL.

A cópia de trabalho, se o destino for um caminho em uma WC.

Acessa o Repositório

Se a origem ou destino da cópia for o repositório, ou se for preciso buscar um dado número de revisão da origem.

Opções

```
--message (-m) TEXT
--file (-F) FILE
--revision (-r) REV
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--force-log
--editor-cmd EDITOR
--encoding ENC
```

--config-dir DIR

Exemplos

Copia um item para dentro de sua cópia de trabalho (apenas agenda a cópia—nada ocorre com o repositório até que você submeta a alteração):

```
$ svn copy foo.txt bar.txt
A         bar.txt
$ svn status
A  +   bar.txt
```

Copia um item de sua cópia de trabalho para uma URL no repositório (com submissão imediata, então você deve informar também uma mensagem de log):

```
$ svn copy near.txt file:///tmp/repos/test/far-away.txt -m "Remote copy."

Committed revision 8.
```

Copia um item do repositório para sua cópia de trabalho (apenas agenda a cópia—nada ocorre com o repositório até que você submeta a alteração):

Esta é a maneira recomendada para ressuscitar um arquivo morto em seu repositório!

```
$ svn copy file:///tmp/repos/test/far-away near-here
A         near-here
```

E, finalmente, copiando entre duas URLs:

```
$ svn copy file:///tmp/repos/test/far-away file:///tmp/repos/test/over-there -m "remote

Committed revision 9.
```

Esta é a forma mais fácil de “rotular” (tag) uma revisão em seu repositório—apenas execute um **svn copy** daquela revisão (normalmente a HEAD) para dentro de seu diretório tags.

```
$ svn copy file:///tmp/repos/test/trunk file:///tmp/repos/test/tags/0.6.32-prerelease -

Committed revision 12.
```

E não se preocupe se você se esquecer de rotular— você pode sempre especificar uma revisão mais antiga e o rótulo a qualquer momento:

```
$ svn copy -r 11 file:///tmp/repos/test/trunk file:///tmp/repos/test/tags/0.6.32-prerel

Committed revision 13.
```

Nome

svn delete — Exclui um item de uma cópia de trabalho ou do repositório.

Sinopse

```
svn delete PATH...
```

```
svn delete URL...
```

Descrição

Os itens especificados pelo *PATH* são agendados para exclusão até o próximo commit. Os arquivos (e diretórios que ainda não foram submetidos) são imediatamente removidos da cópia de trabalho. O comando não irá remover quaisquer itens não versionados ou modificados; utilize a opção `--force` para sobrescrever este comportamento.

Itens especificados pela URL são excluídos do repositório por meio de um registro (commit) imediato. Múltiplas URLs sofrem as alterações de forma atômica.

Nomes Alternativos

del, remove, rm

Altera

Cópia de trabalho, se executando sobre arquivos; repositório, se operando sobre URLs

Acessa o Repositório

Apenas quando executado sobre URLs

Opções

```
--force
--force-log
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--targets FILENAME
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--config-dir DIR
```

Exemplos

Usar **svn** para excluir um arquivo de sua cópia de trabalho remove sua cópia local do arquivo, mas apenas faz o agendamento para que ele seja removido do repositório. Quando você submeter a alteração, o arquivo é excluído efetivamente do repositório.

```
$ svn delete myfile
D         myfile
```

```
$ svn commit -m "Deleted file 'myfile'."
Deleting       myfile
Transmitting file data .
Committed revision 14.
```

Excluir uma URL, no entanto, é imediato, então você deve informar também uma mensagem de log:

```
$ svn delete -m "Deleting file 'yourfile'" file:///tmp/repos/test/yourfile

Committed revision 15.
```

Aqui está um exemplo de como forçar a exclusão de um arquivo que já foi modificado localmente:

```
$ svn delete over-there
svn: Attempting restricted operation for modified resource
svn: Use --force to override this restriction
svn: 'over-there' has local modifications

$ svn delete --force over-there
D         over-there
```

Nome

svn diff — Exibe as diferenças entre duas revisões ou caminhos.

Sinopse

```
diff [-c M | -r N[:M]] [TARGET[@REV]...]
```

```
diff [-r N[:M]] --old=OLD-TGT[@OLDREV] [--new=NEW-TGT[@NEWREV]] [PATH...]
```

```
diff OLD-URL[@OLDREV] NEW-URL[@NEWREV]
```

Descrição

Exibe as diferenças entre dois caminhos. As formas de se usar o **svn diff** são:

Use somente **svn diff'** para exibir as modificações locais em uma cópia de trabalho.

Exibe as mudanças feitas nos *TARGET*s de como eram em *REV* entre duas revisões. *TARGET*s podem ser todos cópias de trabalho como *URL*s. Se *TARGET*s são caminhos na cópia de trabalho, *N* se referirá por padrão a `BASE` e *M*, à cópia de trabalho; se forem *URL*s, *N* deve ser especificado e *M* servirá de referência a `HEAD`. A opção “-c M” é equivalente a “-r N:M” sendo `N = M-1`. Usar “-c -M” faz o contrário: “-r M:N” sendo `N = M-1`.

Exibe as diferenças entre *OLD-TGT* de como estava em *OLDREV* e *NEW-TGT* de como estava em *NEWREV*. *PATH*s, se especificados, são relativos a *OLD-TGT* e *NEW-TGT* e restringem a saída às diferenças entre estes caminhos. *OLD-TGT* e *NEW-TGT* podem ser caminhos na cópia de trabalho ou *URL[@REV]*. *NEW-TGT*, por padrão, se refere a *OLD-TGT* se não for especificada. “-r N” faz com que OLDREV seja N, -r N:M faz com que *OLDREV* seja *N* e que *NEWREV* seja *M*.

Abreviação para **svn diff --old=OLD-URL[@OLDREV] --new=NEW-URL[@NEWREV]**

svn diff -r N:M URL é abreviação para **svn diff -r N:M --old=URL --new=URL**.

svn diff [-r N[:M]] URL1[@N] URL2[@M] é abreviação para **svn diff [-r N[:M]] --old=URL1 --new=URL2**.

Se *TARGET* for uma URL, então as revisões N e M podem ser dadas tanto pela opção `--revision` como pela notação “@” como descrito anteriormente.

Se *TARGET* é um caminho na cópia de trabalho, então a opção `--revision` significa:

`--revision N:M`
: O servidor compara *TARGET@N* e *TARGET@M*.

`--revision N`
: O cliente compara *TARGET@N* com a cópia de trabalho.

(sem `--revision`)
: O cliente compara a base com a cópia de trabalho de *TARGET*.

Se a sintaxe alternativa for usada, o servidor compara *URL1* e *URL2* nas revisões *N* e *M* respectivamente. Se tanto *N* ou *M* forem omitidos, é assumido o valor de `HEAD`.

Por padrão, **svn diff** ignora os diretórios superiores dos caminhos dos arquivos e meramente compara o conteúdo dos dois arquivos sendo comparados. Se você usar `--notice-ancestry`, estes ancestrais dos caminhos serão levados em consideração ao comparar as revisões (isto é, se você executar **svn diff** em dois arquivos com conteúdos idênticos mas com diretórios superiores em caminhos diferentes, você irá ver o conteúdo inteiro do arquivo como se ele tivesse sido removido e adicionado novamente).

Nomes Alternativos

di

Altera

Nada

Acessa o Repositório

Para obter as diferenças de qualquer coisa com relação à revisão BASE em sua cópia de trabalho

Opções

```
--revision (-r) ARG
--change (-c) ARG
--old ARG
--new ARG
--non-recursive (-N)
--diff-cmd CMD
--extensions (-x) "ARGS"
--no-diff-deleted
--notice-ancestry
--summarize
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Exemplos

Compara BASE e sua cópia de trabalho (um dos usos mais comuns do **svn diff**):

```
$ svn diff COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 4404)
+++ COMMITTERS (working copy)
```

Ver o que mudou no arquivo `COMMITTERS` na revisão 9115:

```
$ svn diff -c 9115 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
```

Verificar como as modificações em sua cópia de trabalho se comparam com relação a revisões mais antigas:

```
$ svn diff -r 3900 COMMITTERS
Index: COMMITTERS
```

```
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
```

Comparar a revisão 3000 com a revisão 3500 usando a sintaxe de “@”:

```
$ svn diff http://svn.collab.net/repos/svn/trunk/COMMITTERS@3000 http://svn.collab.net/
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
…
```

Compara a revisão 3000 com a revisão 3500 usando a notação de intervalo (você passa apenas uma URL neste caso):

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk/COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

Compara a revisão 3000 com a revisão 3500 de todos os arquivos em `trunk` usando a notação de intervalo:

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk
```

Compara a revisão 3000 com a revisão 3500 apenas de três arquivos em `trunk` usando a notação de intervalo:

```
$ svn diff -r 3000:3500 --old http://svn.collab.net/repos/svn/trunk COMMITTERS README H
```

Se você já tem uma cópia de trabalho, você pode obter as diferenças sem digitar URLs longas:

```
$ svn diff -r 3000:3500 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

Use `--diff-cmd` *`CMD`* `-x` para passar argumentos diretamente para o programa diff externo

```
$ svn diff --diff-cmd /usr/bin/diff -x "-i -b" COMMITTERS
Index: COMMITTERS
===================================================================
0a1,2
> This is a test
>
```

Nome

svn export — Exporta uma árvore de diretórios limpa.

Sinopse

```
svn export [-r REV] URL[@PEGREV] [PATH]
```

```
svn export [-r REV] PATH1[@PEGREV] [PATH2]
```

Descrição

A primeira forma exporta uma árvore de diretórios limpa a partir do repositório especificado pela URL, na revisão *REV* se esta for dada, ou de `HEAD` em caso contrário, para o caminho dado por *PATH*. Se *PATH* for omitido, o último componente da *URL* é usado como nome do diretório local.

A segunda forma exporta uma árvore de diretórios limpa a partir da cópia de trabalho especificada por *PATH1* para o caminho dado por *PATH2*. Todas as modificações locais serão preservadas, mas arquivos que não estiverem sob controle de versão não serão copiados.

Nomes Alternativos

Nenhum

Altera

Disco local

Acessa o Repositório

Apenas se exportando a partir de uma URL

Opções

```
--revision (-r) REV
--quiet (-q)
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--non-recursive (-N)
--config-dir DIR
--native-eol EOL
--ignore-externals
```

Exemplos

Exporta a partir de sua cópia de trabalho (não exibe nenhum arquivo ou diretório):

```
$ svn export a-wc my-export
Export complete.
```

Exporta diretamente a partir do repositório (exibe cada arquivo e diretório):

```
$ svn export file:///tmp/repos my-export
```

```
A  my-export/test
A  my-export/quiz
…
Exported revision 15.
```

Ao distribuir pacotes de seus repositório específicos para um determinado sistema operacional, pode ser útil exportar uma árvore que use o marcador de fim de linha (EOL) específico. A opção `--native-eol` fará isso, mas ela afeta apenas os arquivos que tenham a propriedade `svn:eol-style = native` anexada a si. Por exemplo, para exportar uma árvore com todas as terminações de linhas dadas do tipo CRLF (possivelmente para distribuição de arquivos .zip para Windows):

```
$ svn export file:///tmp/repos my-export --native-eol CRLF
A  my-export/test
A  my-export/quiz
…
Exported revision 15.
```

Você pode especificar `LR`, `CR`, ou `CRLF` como tipos de marcadores de fim de linha com esta opção `--native-eol`.

Nome

svn help — Ajuda!

Sinopse

```
svn help [SUBCOMMAND...]
```

Descrição

Este é seu melhor amigo quando estiver usando o Subversion e este livro sequer chegará a seus pés!

Nomes Alternativos

?, h

As opções `-?`, `-h` e `--help` têm todas o mesmo efeito que usando o subcomando **help**.

Altera

Nada

Acessa o Repositório

Não

Opções

```
--config-dir DIR
```

Nome

svn import — Submete (*commit*) um arquivo ou árvore não-versionada ao repositório.

Sinopse

```
svn import [PATH] URL
```

Descrição

Submete recursivamente uma cópia de *PATH* para *URL*. Se *PATH* for omitido, então "." é assumido. Diretórios anteriores são criados no repositório, se necessário.

Nomes Alternativos

Nenhum

Altera

Repositório

Acessa o Repositório

Sim

Opções

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--non-recursive (-N)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--force-log
--editor-cmd EDITOR
--encoding ENC
--config-dir DIR
--auto-props
--no-auto-props
--ignore-externals
```

Exemplos

Isto importa o diretório local `myproj` para dentro de `trunk/misc` em seu repositório. O diretório `trunk/misc` não precisa existir antes da operação—**svn import** irá criar diretórios recursivamente para você.

```
$ svn import -m "New import" myproj http://svn.red-bean.com/repos/trunk/misc
Adding         myproj/sample.txt
…
Transmitting file data .........
Committed revision 16.
```

Atente que isto *não* cria um diretório chamado `myproj` no repositório. Se é isso o que você quer, simplesmente adicione `myproj` ao final da URL:

```
$ svn import -m "New import" myproj http://svn.red-bean.com/repos/trunk/misc/myproj
Adding         myproj/sample.txt
…
Transmitting file data .........
Committed revision 16.
```

Depois de importar dados, perceba que a árvore original *not* está sob controle de versão. Para começar a trabalhar, você ainda precisa realizar um **svn checkout** para obter uma nova cópia de trabalho da árvore.

Nome

svn info — Exibe informação sobre um item local ou remoto.

Sinopse

```
svn info [TARGET[@REV]...]
```

Descrição

Exibe informação sobre caminhos na cópia de trabalho ou URLs especificadas. A informação exibida para ambos pode incluir:

- Path
- Name
- URL
- Repository Root
- Repository UUID
- Revision
- Node Kind
- Last Changed Author
- Last Changed Revision
- Last Changed Date
- Lock Token
- Lock Owner
- Lock Created (date)
- Lock Expires (date)

Tipos adicionais de informação disponíveis apenas para caminhos em cópias de trabalho são:

- Schedule
- Copied From URL
- Copied From Rev
- Text Last Updated
- Properties Last Updated
- Checksum
- Conflict Previous Base File
- Conflict Previous Working File
- Conflict Current Base File
- Conflict Properties File

Nomes Alternativos

Nenhum

Altera

Nada

Acessa o Repositório

Apenas se operando sobre URLs

Opções

```
--revision (-r) REV
--recursive (-R)
--targets FILENAME
--incremental
--xml
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Exemplos

svn info irá mostrar a você toda a informação útil que tiver para itens em sua cópia de trabalho. Ele exibirá informações sobre arquivos:

```
$ svn info foo.c
Path: foo.c
Name: foo.c
URL: http://svn.red-bean.com/repos/test/foo.c
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 4417
Node Kind: file
Schedule: normal
Last Changed Author: sally
Last Changed Rev: 20
Last Changed Date: 2003-01-13 16:43:13 -0600 (Mon, 13 Jan 2003)
Text Last Updated: 2003-01-16 21:18:16 -0600 (Thu, 16 Jan 2003)
Properties Last Updated: 2003-01-13 21:50:19 -0600 (Mon, 13 Jan 2003)
Checksum: d6aeb60b0662ccceb6bce4bac344cb66
```

Ele também exibirá informações sobre diretórios:

```
$ svn info vendors
Path: vendors
URL: http://svn.red-bean.com/repos/test/vendors
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 19
Node Kind: directory
Schedule: normal
```

```
Last Changed Author: harry
Last Changed Rev: 19
Last Changed Date: 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003)
Properties Last Updated: 2003-01-16 23:39:02 -0600 (Thu, 16 Jan 2003)
```

svn info também funciona em URLs (perceba também que o arquivo readme.doc neste exemplo está travado, então a informação da trava também é exibida):

```
$ svn info http://svn.red-bean.com/repos/test/readme.doc
Path: readme.doc
Name: readme.doc
URL: http://svn.red-bean.com/repos/test/readme.doc
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 1
Node Kind: file
Schedule: normal
Last Changed Author: sally
Last Changed Rev: 42
Last Changed Date: 2003-01-14 23:21:19 -0600 (Tue, 14 Jan 2003)
Lock Token: opaquelocktoken:14011d4b-54fb-0310-8541-dbd16bd471b2
Lock Owner: harry
Lock Created: 2003-01-15 17:35:12 -0600 (Wed, 15 Jan 2003)
Lock Comment (1 line):
My test lock comment
```

Nome

svn list — Lista entradas de diretório no repositório.

Sinopse

```
svn list [TARGET[@REV]...]
```

Descrição

Lista cada arquivo em *`TARGET`* e o conteúdo de cada diretório em *`TARGET`* como existirem no repositório. Se *`TARGET`* for um caminho em uma cópia de trabalho, a URL correspondente no repositório será usada.

O *`TARGET`* padrão, quando não informado, é “`.`”, significando a URL correspondente ao diretório atual na cópia de trabalho.

Com `--verbose`, **svn list** exibe os seguintes campos para cada item:

- Número da revisão do último commit
- Autor do último commit
- Se travado, a letra “O” (Veja svn info para detalhes).
- Tamanho (em bytes)
- Data e horário do último commit

Com `--xml`, a saída fica em um formato XML (com um cabeçalho e um elemento de documento encapsulador, a menos que `--incremental` também seja especificado). Toda a informação está presente; a opção `--verbose` não é aceita.

Nomes Alternativos

ls

Altera

Nada

Acessa o Repositório

Sim

Opções

```
--revision (-r) REV
--verbose (-v)
--recursive (-R)
--incremental
--xml
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Exemplos

svn list é mais útil se você quiser ver que arquivos um repositório tem sem precisar obter uma cópia de trabalho:

```
$ svn list http://svn.red-bean.com/repos/test/support
README.txt
INSTALL
examples/
…
```

Você pode passar a opção `--verbose` para informações adicionais, similar ao comando UNIX **ls -l**:

```
$ svn list --verbose file:///tmp/repos
     16 sally         28361 Jan 16 23:18 README.txt
     27 sally             0 Jan 18 15:27 INSTALL
     24 harry               Jan 18 11:27 examples/
```

Para mais detalhes, veja “svn list”.

Nome

svn lock — Trava caminhos na cópia de trabalho ou em URLs no repositório, assim nenhum outro usuário poderá submeter alterações neles.

Sinopse

```
svn lock TARGET...
```

Descrição

Trava cada *TARGET*. Se algum *TARGET* já estiver travado por outro usuário, exibe uma mensagem de aviso e continua travando os demais *TARGET*s informados. Use `--force` para roubar uma trava de outro usuário ou cópia de trabalho.

Nomes Alternativos

Nenhum

Altera

Cópia de trabalho, Repositório

Acessa o Repositório

Sim

Opções

```
--targets FILENAME
--message (-m) TEXT
--file (-F) FILE
--force-log
--encoding ENC
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--force
```

Exemplos

Trava dois arquivos em sua cópia de trabalho:

```
$ svn lock tree.jpg house.jpg
'tree.jpg' locked by user 'harry'.
'house.jpg' locked by user 'harry'.
```

Trava um arquivo em sua cópia de trabalho que já estava travado por outro usuário:

```
$ svn lock tree.jpg
svn: warning: Path '/tree.jpg is already locked by user 'sally in \
     filesystem '/svn/repos/db'
```

```
$ svn lock --force tree.jpg
'tree.jpg' locked by user 'harry'.
```

Trava um arquivo sem uma cópia de trabalho:

```
$ svn lock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg' locked by user 'harry'.
```

Para mais detalhes, veja “Travamento”.

Nome

svn log — Exibe as mensagens de log submetidas.

Sinopse

```
svn log [PATH]
```

```
svn log URL [PATH...]
```

```
svn log URL[@REV] [PATH...]
```

Descrição

Exibe mensagens de log do repositório. Se nenhum argumento for informado, **svn log** exibe as mensagens de log para todos os arquivos e diretórios dentro do diretório atual (inclusive) de sua cópia de trabalho. Você pode refinar os resultados especificando um caminho, uma ou mais revisões, ou qualquer combinação das duas. O intervalo padrão de revisões para caminhos locais é `BASE:1`.

Se você especificar uma URL sozinha, então o comando exibe as mensagens de log para tudo o que a URL contiver. Se você adicionar caminhos depois da URL, apenas as mensagens para estes caminhos sob a URL serão exibidas. O intervalo padrão de revisões para uma URL é `HEAD:1`.

Com `--verbose`, **svn log** também irá exibir todos os caminhos afetados com cada mensagem de log. Com `--quiet`, **svn log** não irá exibir o corpo da mensagem de log em si (isto é compatível com `--verbose`).

Cada mensagem de log é exibida apenas uma vez, mesmo se mais de um dos caminhos afetados por aquela revisão forem solicitados explicitamente. As mensagens de log seguem o histórico de cópias por padrão. Use `--stop-on-copy` para desabilitar este comportamento, o que pode ser útil para determinar pontos de ramificação.

Nomes Alternativos

Nenhum

Altera

Nada

Acessa o Repositório

Sim

Opções

```
--revision (-r) REV
--quiet (-q)
--verbose (-v)
--targets FILENAME
--stop-on-copy
--incremental
--limit NUM
--xml
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Exemplos

Você pode ver as mensagens de log para todos os caminhos que sofreram alterações em sua cópia de trabalho, executando **svn log** a partir da raiz da mesma:

```
$ svn log
------------------------------------------------------------------------
r20 | harry | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Tweak.
------------------------------------------------------------------------
r17 | sally | 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003) | 2 lines
…
```

Examine todas as mensagens de log para um dado arquivo em sua cópia de trabalho:

```
$ svn log foo.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

Se você não tiver uma cópia de trabalho à mão, você pode obter as mensagens de log a partir de uma URL:

```
$ svn log http://svn.red-bean.com/repos/test/foo.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

Se você quiser as mensagens de diversos arquivos sob uma mesma URL, você pode usar a sintaxe `URL [PATH...]`.

```
$ svn log http://svn.red-bean.com/repos/test/ foo.c bar.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r31 | harry | 2003-01-10 12:25:08 -0600 (Fri, 10 Jan 2003) | 1 line

Added new file bar.c
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

Quando estiver concatenando os resultados de múltiplas chamadas ao comando log, você também pode usar a opção `--incremental`. O **svn log** normalmente exibe uma linha tracejada no começo de

cada mensagem de log, depois de cada mensagem de log subsequente, e após a última mensagem de log. Se você executou o **svn log** com um intervalo de revisões, você deve ter obtido isto:

```
$ svn log -r 14:15
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
r15 | …

------------------------------------------------------------------------
```

Entretanto, se você queria gravar duas mensagens de log não-sequenciais em um arquivo, você poderia ter feito algo assim:

```
$ svn log -r 14 > mylog
$ svn log -r 19 >> mylog
$ svn log -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
------------------------------------------------------------------------
r19 | …

------------------------------------------------------------------------
------------------------------------------------------------------------
r27 | …

------------------------------------------------------------------------
```

Você pode evitar a bagunça das duplas linhas tracejadas em sua saída usando a opção incremental:

```
$ svn log --incremental -r 14 > mylog
$ svn log --incremental -r 19 >> mylog
$ svn log --incremental -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
r19 | …

------------------------------------------------------------------------
r27 | …
```

A opção `--incremental` resulta em um controle da saída semelhante também quando usada com a opção `--xml`.

Se você executar **svn log** em um dado específico e informar uma revisão específica e não obtiver nada como saída

```
$ svn log -r 20 http://svn.red-bean.com/untouched.txt
------------------------------------------------------------------------
```

Isto apenas significa que aquele caminho não foi modificado naquela revisão. Se você obtiver o log a partir da raiz do repositório, ou souber o arquivo que foi alterado naquela revisão, você pode especificá-lo explicitamente:

```
$ svn log -r 20 touched.txt
------------------------------------------------------------------------
r20 | sally | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Made a change.
------------------------------------------------------------------------
```

Nome

svn merge — Aplica as diferenças entre duas fontes para um caminho numa cópia de trabalho.

Sinopse

```
svn merge [-c M | -r N:M] SOURCE[@REV] [WCPATH]
```

```
svn merge sourceURL1[@N] sourceURL2[@M] [WCPATH]
```

```
svn merge sourceWCPATH1@N sourceWCPATH2@M [WCPATH]
```

Descrição

Na primeira e na segunda formas, os caminhos de origem (URLs na primeira forma, caminhos na cópia de trabalho na segunda) são especificados nas revisões *N* e *M*. Estas são as duas fontes a serem comparadas. A revisão padrão, quando omitida, será `HEAD`.

A opção `-c M` é equivalente a `-r N:M` onde *N* = *M-1*. Usar `-c -M` faz o inverso: `-r M:N` onde *N* = *M-1*.

Na terceira forma, *SOURCE* pode ser uma URL ou um item numa cópia de trabalho, neste caso a URL correspondente é usada. Esta URL, nas revisões *N* e *M*, define as duas fontes a serem comparadas.

WCPATH é o caminho na cópia de trabalho que irão receber as alterações. Se *WCPATH* for omitido, será assumido "`.`" como valor padrão, a menos que as fontes tenham nomes de base idênticos que correspondam a um arquivo dentro de "`.`": neste caso, as diferenças serão aplicadas àquele arquivo.

Diferentemente do **svn diff**, o comando merge leva os diretórios anteriores de um arquivo em consideração ao executar uma operação de mesclagem. Isto é muito importante quando você estiver juntando as alterações feitas de um ramo para outro e tiver renomeado um arquivo em um ramo mas não em outro.

Nomes Alternativos

Nenhum

Altera

Cópia de trabalho

Acessa o Repositório

Apenas se trabalhando com URLs

Opções

```
--revision (-r) REV
--change (-c) REV
--non-recursive (-N)
--quiet (-q)
--force
--dry-run
--diff3-cmd CMD
--extensions (-x) ARG
--ignore-ancestry
--username USER
--password PASS
--no-auth-cache
```

```
--non-interactive
--config-dir DIR
```

Exemplos

Mescla um ramo de volta ao trunk (assumindo que você tem uma cópia de trabalho de trunk, e que o ramo tenha sido criado na revisão 250):

```
$ svn merge -r 250:HEAD http://svn.red-bean.com/repos/branches/my-branch
U  myproj/tiny.txt
U  myproj/thhgttg.txt
U  myproj/win.txt
U  myproj/flo.txt
```

Se você tiver feito a ramificação na revisão 23, e quiser mesclar as alterações feitas no trunk para seu ramo, você pode fazer isto de dentro da cópia de trabalho de seu ramo:

```
$ svn merge -r 23:30 file:///tmp/repos/trunk/vendors
U  myproj/thhgttg.txt
…
```

Para mesclar as alterações em um único arquivo:

```
$ cd myproj
$ svn merge -r 30:31 thhgttg.txt
U  thhgttg.txt
```

Nome

svn mkdir — Cria um novo diretório sob controle de versão.

Sinopse

```
svn mkdir PATH...
```

```
svn mkdir URL...
```

Descrição

Cria um diretório com o nome dado pelo último componente de *PATH* ou da URL. Um diretório especificado por *PATH* como um caminho na cópia de trabalho é agendado para adição. Um diretório especificado por uma URL é criado no repositório por meio de um commit imediato. Múltiplas URLs de diretório são submetidas atomicamente. Em ambos os casos, todos os diretórios intermediários já devem existir.

Nomes Alternativos

Nenhum

Altera

Cópia de trabalho, repositório se executando sobre uma URL

Acessa o Repositório

Apenas se executando sobre uma URL

Opções

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--force-log
--config-dir DIR
```

Exemplos

Cria um diretório em sua cópia de trabalho:

```
$ svn mkdir newdir
A         newdir
```

Cria um diretório no repositório (submissão instantânea, então uma mensagem de log é requerida):

```
$ svn mkdir -m "Making a new dir." http://svn.red-bean.com/repos/newdir

Committed revision 26.
```

Nome

svn move — Move um arquivo ou diretório.

Sinopse

```
svn move SRC DST
```

Descrição

Este comando move um arquivo ou diretório em sua cópia de trabalho ou no repositório.

Este comando é equivalente a um **svn copy** seguido de um **svn delete**.

O Subversion não movimentação entre cópias de trabalho e URLs. Além disso, você só pode mover arquivos dentro de um único repositório—o Subversion não suporta movimentação entre repositórios.

WC -> WC
: Move e agenda um arquivo ou diretório para adição (com histórico).

URL -> URL
: Renomeação inteiramente no lado do servidor.

Nomes Alternativos

mv, rename, ren

Altera

Cópia de trabalho, repositório se executando sobre uma URL

Acessa o Repositório

Apenas se executando sobre uma URL

Opções

```
--message (-m) TEXT
--file (-F) FILE
--revision (-r) REV (Obsoleto)
--quiet (-q)
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--force-log
--config-dir DIR
```

Exemplos

Move um arquivo em sua cópia de trabalho:

```
$ svn move foo.c bar.c
A         bar.c
D         foo.c
```

Move um arquivo no repositório (commit imediato, necessitando, então, de uma mensagem de log):

```
$ svn move -m "Move a file" http://svn.red-bean.com/repos/foo.c \
                            http://svn.red-bean.com/repos/bar.c

Committed revision 27.
```

Nome

svn propdel — Remove uma propriedade de um item.

Sinopse

```
svn propdel PROPNAME [PATH...]
```

```
svn propdel PROPNAME --revprop -r REV [TARGET]
```

Descrição

Este comando remove as propriedades de arquivos, diretórios, ou revisões. A primeira forma remove propriedades versionadas em sua cópia de trabalho, enquanto que a segunda remove propriedades não-versionadas remotas em uma revisão no repositório (*TARGET* apenas determina qual repositório acessar).

Nomes Alternativos

pdel, pd

Altera

Cópia de trabalho, repositório apenas se executando sobre uma URL

Acessa o Repositório

Apenas se executando sobre uma URL

Opções

```
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Exemplos

Remove uma propriedade de um arquivo em sua cópia de trabalho

```
$ svn propdel svn:mime-type  some-script
property 'svn:mime-type' deleted from 'some-script'.
```

Remove uma propriedade de uma revisão:

```
$ svn propdel --revprop -r 26 release-date
property 'release-date' deleted from repository revision '26'
```

Nome

svn propedit — Edita a propriedade de um ou mais itens sob controle de versão.

Sinopse

```
svn propedit PROPNAME PATH...
```

```
svn propedit PROPNAME --revprop -r REV [TARGET]
```

Descrição

Edita uma ou mais propriedades usando seu editor preferido. A primeira forma edita propriedades versionadas em sua cópia de trabalho, enquanto que a segunda edita propriedades não-versionadas remotas em uma revisão de repositório (*TARGET* apenas determina qual repositório acessar).

Nomes Alternativos

pedit, pe

Altera

Cópia de trabalho, repositório apenas se executando sobre uma URL

Acessa o Repositório

Apenas se executando sobre uma URL

Opções

```
--revision (-r) REV
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--editor-cmd EDITOR
--config-dir DIR
```

Exemplos

svn propedit facilita a modificação de propriedades que tenham múltiplos valores:

```
$ svn propedit svn:keywords  foo.c
    <svn will launch your favorite editor here, with a buffer open
    containing the current contents of the svn:keywords property.  You
    can add multiple values to a property easily here by entering one
    value per line.>
Set new value for property 'svn:keywords' on 'foo.c'
```

Nome

svn propget — Exibe o valor de uma propriedade.

Sinopse

```
svn propget PROPNAME [TARGET[@REV]...]
```

```
svn propget PROPNAME --revprop -r REV [URL]
```

Descrição

Exibe o valor de uma propriedade de arquivos, diretórios ou revisões. A primeira forma exibe a propriedade versionada de um item ou itens em sua cópia de trabalho, enquanto que a segunda exibe propriedades remotas não-versionadas em uma revisão de repositório. Veja “Propriedades” para mais informações sobre propriedades.

Nomes Alternativos

pget, pg

Altera

Cópia de trabalho, repositório apenas se executando sobre uma URL

Acessa o Repositório

Apenas se executando sobre uma URL

Opções

```
--recursive (-R)
--revision (-r) REV
--revprop
--strict
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Exemplos

Examina uma propriedade de um arquivo em sua cópia de trabalho:

```
$ svn propget svn:keywords foo.c
Author
Date
Rev
```

O mesmo para revisões de propriedades:

```
$ svn propget svn:log --revprop -r 20
Began journal.
```

Nome

svn proplist — Lista todas as propriedades.

Sinopse

```
svn proplist [TARGET[@REV]...]
```

```
svn proplist --revprop -r REV [TARGET]
```

Descrição

Lista todas as propriedades de arquivos, diretórios, ou revisões. A primeira forma lista propriedades versionadas em sua cópia de trabalho, enquanto que a segunda lista propriedades remotas não-versionadas em uma revisão de repositório (*TARGET* apenas determina que repositório acessar).

Nomes Alternativos

plist, pl

Altera

Cópia de trabalho, repositório apenas se executando sobre uma URL

Acessa o Repositório

Apenas se executando sobre uma URL

Opções

```
--verbose (-v)
--recursive (-R)
--revision (-r) REV
--quiet (-q)
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Exemplos

Você pode usar proplist para ver as propriedades de um item em sua cópia de trabalho:

```
$ svn proplist foo.c
Properties on 'foo.c':
  svn:mime-type
  svn:keywords
  owner
```

Mas com a opção `--verbose`, o svn proplist torna-se extremamente útil pois também lhe mostra os valores das propriedades:

```
$ svn proplist --verbose foo.c
Properties on 'foo.c':
```

```
svn:mime-type : text/plain
svn:keywords : Author Date Rev
owner : sally
```

Nome

svn propset — Define PROPNAME para PROPVAL em arquivos, diretórios, ou revisões.

Sinopse

```
svn propset PROPNAME [PROPVAL | -F VALFILE] PATH...
```

```
svn propset PROPNAME --revprop -r REV [PROPVAL | -F VALFILE] [TARGET]
```

Descrição

Define *PROPNAME* para *PROPVAL* em arquivos, diretórios, ou revisões. O primeiro exemplo cria uma modificação numa propriedade local, versionada, na cópia de trabalho; e o segundo cria uma modificação numa propriedade remota, não-versionada, em uma revisão de repositório (*TARGET* apenas determina que repositório acessar).

O Subversion tem um conjunto de propriedades “especiais” que afetam seu comportamento. Veja “Subversion properties” para mais sobre estas propriedades.

Nomes Alternativos

pset, ps

Altera

Cópia de trabalho, repositório apenas se executando sobre uma URL

Acessa o Repositório

Apenas se executando sobre uma URL

Opções

```
--file (-F) FILE
--quiet (-q)
--revision (-r) REV
--targets FILENAME
--recursive (-R)
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--force
--config-dir DIR
```

Exemplos

Define o tipo mime de um arquivo:

```
$ svn propset svn:mime-type image/jpeg foo.jpg
property 'svn:mime-type' set on 'foo.jpg'
```

Em um sistema UNIX, se você quiser que um arquivo tenha a permissão de execução definida:

```
$ svn propset svn:executable ON somescript
property 'svn:executable' set on 'somescript'
```

Talvez você tenha uma política interna que defina certas propriedades para beneficiar seus colaboradores:

```
$ svn propset owner sally foo.c
property 'owner' set on 'foo.c'
```

Se você se enganar escrevendo uma mensagem de log para uma dada revisão e quiser mudá-la, use `--revprop` e atribua o valor de `svn:log` para a nova mensagem de log:

```
$ svn propset --revprop -r 25 svn:log "Journaled about trip to New York."
property 'svn:log' set on repository revision '25'
```

Ou, se você não tiver uma cópia de trabalho, você pode informar uma URL.

```
$ svn propset --revprop -r 26 svn:log "Document nap." http://svn.red-bean.com/repos
property 'svn:log' set on repository revision '25'
```

Por fim, você pode informar ao propset para obter suas entradas a partir de um arquivo. Você ainda pode usar isso para definir o conteúdo de uma propriedade para qualquer valor binário:

```
$ svn propset owner-pic -F sally.jpg moo.c
property 'owner-pic' set on 'moo.c'
```

Por padrão, você não pode modificar propriedades de revisão em um repositório Subversion. O administrador de seu repositório deve explicitamente permitir modificações em revisões de propriedades criando um script de extensão (*hook*) chamado `pre-revprop-change`. Veja “Implementing Repository Hooks” para mais informações sobre scripts de extensão.

Nome

svn resolved — Remove arquivo ou diretórios da cópia de trabalho do estado de “conflito”.

Sinopse

```
svn resolved PATH...
```

Descrição

Remove o estado de “conflito” em arquivos ou diretórios numa cópia de trabalho. Esta rotina não resolve marcadores de conflito semanticamente; ele apenas remove os arquivos auxiliares relacionados ao conflito e permite que o *PATH* possa ser submetido novamente; isto é, ele informa ao Subversion que o conflito foi “resolvido”. Leia “Resolvendo Conflitos (Combinando Alterações de Outros)” para uma visão mais aprofundada sobre resolução de conflitos.

Nomes Alternativos

Nenhum

Altera

Cópia de trabalho

Acessa o Repositório

Não

Opções

```
--targets FILENAME
--recursive (-R)
--quiet (-q)
--config-dir DIR
```

Exemplos

Se você obtiver um conflito em uma atualização, sua cópia de trabalho irá gerar três novos arquivos:

```
$ svn update
C  foo.c
Updated to revision 31.
$ ls
foo.c
foo.c.mine
foo.c.r30
foo.c.r31
```

Uma vez que você tenha resolvido o conflito e que `foo.c` esteja pronto para ser submetido, execute o comando **svn resolved** para fazer com que sua cópia de trabalho saiba que você já cuidou de tudo.

Você *pode* apenas remover os arquivos relacionados ao conflito e realizar a operação de commit, mas o **svn resolved** corrige alguns dados de registro na área administrativa na cópia de trabalho além de remover os arquivos conflituosos, de forma que recomendamos que você utilize este comando.

Nome

svn revert — Desfaz todas as edições locais.

Sinopse

```
svn revert PATH...
```

Descrição

Reverte quaisquer alterações feitas em um arquivo ou diretório e resolve quaisquer estados de conflito. **svn revert** não apenas irá reverter o conteúdo de um item em sua cópia de trabalho mas também quaisquer modificação de propriedades. Finalmente, você pode usá-lo para desfazer qualquer operação de agendamento que você possa ter feito (p.ex., arquivos agendados para adição ou remoção podem ser “desagendados”).

Nomes Alternativos

Nenhum

Altera

Cópia de trabalho

Acessa o Repositório

Não

Opções

```
--targets FILENAME
--recursive (-R)
--quiet (-q)
--config-dir DIR
```

Exemplos

Descarta as alterações em um arquivo:

```
$ svn revert foo.c
Reverted foo.c
```

Se você quiser reverter um diretório inteiro de arquivos, use a opção `--recursive`:

```
$ svn revert --recursive .
Reverted newdir/afile
Reverted foo.c
Reverted bar.txt
```

Por último, você pode desfazer quaisquer operações de agendamento:

```
$ svn add mistake.txt whoops
A         mistake.txt
A         whoops
A         whoops/oopsie.c
```

```
$ svn revert mistake.txt whoops
Reverted mistake.txt
Reverted whoops

$ svn status
?      mistake.txt
?      whoops
```

O **svn revert** intrinsecamente perigoso, já que todo o seu propósito é descartar dados—especificamente, modificações não submetidas ao repositório. Uma vez que você tenha feito uma reversão, o Subversion *não disponibiliza nenhuma forma* de obter estes dados de suas alterações não submetidas.

Se você não informar um alvo para o **svn revert**, ele não fará nada—para protegê-lo de perder suas alterações acidentalmente em sua cópia de trabalho, o **svn revert** requer que você informe pelo menos um alvo.

Nome

svn status — Exibe informação sobre o estado de arquivos e diretórios na cópia de trabalho.

Sinopse

```
svn status [PATH...]
```

Descrição

Exibe informação sobre o estado de arquivos e diretórios na cópia de trabalho. Sem argumentos, o comando exibe apenas os itens modificados localmente (sem acesso ao repositório). Com `--show-updates`, ele adiciona informação sobre a revisão de trabalho e informações defasadas de servidor. Com `--verbose`, exibe informações completas sobre revisão de cada item.

As primeiras seis colunas na saída têm um caractere de largura cada, e cada coluna lhe dá informação sobre diferentes aspectos de cada item na cópia de trabalho.

A primeira coluna indica que um item foi adicionado, removido, ou então alterado.

' '
: Sem modificações.

'A'
: O item foi agendado para adição.

'D'
: O item foi agendado para remoção (deleção).

'M'
: O item está sendo modificado.

'R'
: O item foi substituído em sua cópia de trabalho. Isto significa que o arquivo foi agendado para remoção, e então um novo arquivo com o mesmo nome foi agendado para adição em seu lugar.

'C'
: O conteúdo (ao contrário das propriedades) do item entra em conflito com as atualizações recebidas do repositório.

'X'
: O item está presente devido a uma definição externa.

'I'
: O item está sendo ignorado (p.ex., com a propriedade `svn:ignore`).

'?'
: O item não está sob controle de versão.

'!'
: O item está faltando (p.ex., você o moveu ou removeu sem usar o **svn**). Isto também indica que um diretório está incompleto (uma operação de checkout ou update foi interrompida).

'~'
: O item está versionado como um tipo de objeto (arquivo, diretório, link), mas está sendo substituído por um tipo diferente de objeto.

A segunda coluna indica o estado das propriedades de um arquivo ou diretório.

' '
: Sem modificações.

'M'
: Propriedades deste item foram modificadas.

'C'
: As propriedades para este item entram em conflito com as atualizações de propriedades recebidas do repositório.

A terceira coluna é preenchida apenas se a cópia de trabalho estiver travada. (Veja “Às Vezes Você Só Precisa Limpar”.)

' '
: O item não está travado.

'L'
: O item está travado.

A quarta coluna é preenchida se o item estiver agendado para adição com histórico.

' '
: O histórico não está agendado para submissão.

'+'
: O histórico está agendado para submissão.

A quinta coluna é preenchida apenas se o item tiver sido trocado relativo a seus diretórios anteriores (veja “Atravessando Ramos”).

' '
: O item é filho de seu diretório pai.

'S'
: O item foi trocado.

A sexta coluna contém informação sobre trava.

' '
: Quando `--show-updates` é usado, o arquivo não está travado. Se `--show-updates` *não* é usado, isto apenas significa que o arquivo não está travado nesta cópia de trabalho.

K
: O arquivo está travado nesta cópia de trabalho.

O
: O arquivo ou está travado por outro usuário ou em outra cópia de trabalho. Isto só aparece quando `--show-updates` é usado.

T
: O arquivo estava travado nesta cópia de trabalho, mas a trava foi “roubada” e está inválida. O arquivo atualmente está travado no repositório. Isto só aparece quando `--show-updates` é usado.

B
: O arquivo estava travado nesta cópia de trabalho, mas a trava está sendo “quebrada” e está inválida. O arquivo não está mais travado. Isto só aparece quando `--show-updates` é usado

A informação defasada aparece na sétima coluna (apenas se você passar a opção `--show-updates`).

' '
: O item em sua cópia de trabalho está atualizado.

'*'
: Uma nova revisão do item existe no servidor.

Os campos restantes têm largura variável e são delimitados por espaços. A revisão de trabalho é o próximo campo se as opções `--show-updates` ou `--verbose` forem passadas.

Se a opção `--verbose` for passada, a última revisão submetida e o último autor são exibidos na sequência.

O caminho na cópia de trabalho é sempre o último campo, então ele pode incluir espaços.

Nomes Alternativos

stat, st

Altera

Nada

Acessa o Repositório

Apenas quando usando `--show-updates`

Opções

```
--show-updates (-u)
--verbose (-v)
--non-recursive (-N)
--quiet (-q)
--no-ignore
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--ignore-externals
```

Exemplos

Esta é a forma mais fácil de encontrar quais alterações você fez em sua cópia de trabalho:

```
$ svn status wc
 M     wc/bar.c
A  +   wc/qax.c
```

Se você quiser encontrar quais arquivos em sua cópia de trabalho estão desatualizados, passe a opção `--show-updates` (isto *não* faz quaisquer alterações em sua cópia de trabalho). Aqui você pode ver que `wc/foo.c` foi modificado no repositório desde a última atualização em sua cópia de trabalho:

```
$ svn status --show-updates wc
 M           965    wc/bar.c
      *      965    wc/foo.c
A  +         965    wc/qax.c
Status against revision:    981
```

`--show-updates` *apenas* coloca um asterisco próximo aos itens que estão desatualizados (isto é, os itens que serão atualizados a partir do repositório se você executar um **svn update**). `--show-updates` *não* faz com que a listagem de estado reflita a versão de repositório do item (apesar de que você pode ver o número de revisão do repositório passando a opção `--verbose`).

E finalmente, o máximo de informação que você pode obter com o subcomando status:

```
$ svn status --show-updates --verbose wc
 M           965       938 sally        wc/bar.c
       *     965       922 harry        wc/foo.c
A  +         965       687 harry        wc/qax.c
             965       687 harry        wc/zig.c
Head revision:   981
```

Para mais exemplos de **svn status**, veja “Obtendo uma visão geral de suas alterações”.

Nome

svn switch — Atualiza a cópia de trabalho para uma URL diferente.

Sinopse

```
svn switch URL [PATH]
```

```
switch --relocate FROM TO [PATH...]
```

Descrição

A primeira variante deste subcomando (sem a opção `--relocate`) atualiza sua cópia de trabalho para apontar para uma nova URL—frequentemente uma URL que compartilhe um ancestral comum com sua cópia de trabalho, apesar de não ser necessário. Esta é a forma que o Subversion usa para mover uma cópia de trabalho para um novo ramo. Consulte “Atravessando Ramos” para uma visão mais aprofundada sobre este recurso.

A opção `--relocate` faz com que o **svn switch** execute algo diferente: ele atualiza sua cópia de trabalho para apontar para *o mesmo* diretório no repositório, apenas numa URL diferente (tipicamente porque um administrador moveu o repositório para um outro servidor, ou para outra URL no mesmo servidor).

Nomes Alternativos

sw

Altera

Cópia de trabalho

Acessa o Repositório

Sim

Opções

```
--revision (-r) REV
--non-recursive (-N)
--quiet (-q)
--diff3-cmd CMD
--relocate FROM TO
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Exemplos

Se você atualmente dentro do diretório `vendors`, o qual foi ramificado para `vendors-with-fix`, e você gostaria de trocar sua cópia de trabalho para esse ramo:

```
$ svn switch http://svn.red-bean.com/repos/branches/vendors-with-fix .
U  myproj/foo.txt
U  myproj/bar.txt
U  myproj/baz.c
```

```
U  myproj/qux.c
Updated to revision 31.
```

A para trocar de volta, apenas informa a URL para o local no repositório a partir do qual você originalmente obteve sua cópia de trabalho:

```
$ svn switch http://svn.red-bean.com/repos/trunk/vendors .
U  myproj/foo.txt
U  myproj/bar.txt
U  myproj/baz.c
U  myproj/qux.c
Updated to revision 31.
```

Você pode apenas trocar parte de sua cópia de trabalho para um ramo se você não quiser trocar toda a sua cópia de trabalho.

Algumas vezes um administrador pode mudar a “localização base” de seu repositório—em outras palavras, o conteúdo do repositório não muda, mas a URL principal que dá acesso à raiz do repositório sim. Por exemplo, o nome do host pode mudar, o esquema da URL pode mudar, ou qualquer parte da URL relacionada ao repositório em si pode mudar. Ao invés de obter (*check out*) uma nova cópia de trabalho, você pode fazer com que o comando **svn switch** “reescreva” os começos de todas as URLs em sua cópia de trabalho. Use a opção `--relocate` para fazer a substituição. Nenhum conteúdo de arquivo é modificado, nem sequer o repositório é contactado. É semelhante a executar um script em Perl sobre os diretórios `.svn/` de sua cópia de trabalho, executando **s/OldRoot/NewRoot/**.

```
$ svn checkout file:///tmp/repos test
A  test/a
A  test/b
…

$ mv repos newlocation
$ cd test/

$ svn update
svn: Unable to open an ra_local session to URL
svn: Unable to open repository 'file:///tmp/repos'

$ svn switch --relocate file:///tmp/repos file:///tmp/newlocation .
$ svn update
At revision 3.
```

Tenha cuidado ao usar a opção `--relocate`. Se você errar na digitação do argumento, você pode terminar criando URLs sem sentido relacionadas a sua cópia de trabalho que deixem todo o seu espaço de trabalho inutilizável e difícil de corrigir. Também é importante entender exatamente quando se deve ou não usar `--relocate`. Aqui vai uma regra de ouro:

- Se sua cópia de trabalho precisar refletir um novo diretório *dentro* do repositório, então apenas use **svn switch**.
- Se a cópia de trabalho ainda reflete o mesmo diretório no repositório, mas o local do repositório em si foi mudado, então use **svn switch --relocate**.

Nome

svn unlock — Destrava caminhos na cópia de trabalho ou URLs.

Sinopse

```
svn unlock TARGET...
```

Descrição

Destrava cada *TARGET*. Se algum *TARGET* estiver ou travado por outro usuário ou se nenhum token de travamento válido existir na cópia de trabalho, exibe um aviso e continua destravando os demais *TARGET*s. Use `--force` para quebrar uma trava que pertença a outro usuário na cópia de trabalho.

Nomes Alternativos

Nenhum

Altera

Cópia de trabalho, Repositório

Acessa o Repositório

Sim

Opções

```
--targets FILENAME
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--force
```

Exemplos

Destrava dois arquivos em sua cópia de trabalho:

```
$ svn unlock tree.jpg house.jpg
'tree.jpg' unlocked.
'house.jpg' unlocked.
```

Destrava um arquivo em sua cópia de trabalho que já está travado por outro usuário:

```
$ svn unlock tree.jpg
svn: 'tree.jpg' is not locked in this working copy
$ svn unlock --force tree.jpg
'tree.jpg' unlocked.
```

Destrava um arquivo sem ter uma cópia de trabalho:

```
$ svn unlock http://svn.red-bean.com/repos/test/tree.jpg
```

```
'tree.jpg unlocked.
```

Para mais detalhes, veja “Travamento”.

Nome

svn update — Update your working copy.

Sinopse

```
svn update [PATH...]
```

Descrição

svn update traz as alterações do repositório para sua cópia de trabalho. Se nenhuma revisão for informada, ele atualiza sua cópia de trabalho para a revisão `HEAD`. Do contrário, ele sincroniza a cópia de trabalho para a revisão dada pela opção `--revision`. Como parte da sincronização, o **svn update** também remove quaisquer travas roubadas (veja “Às Vezes Você Só Precisa Limpar”) encontradas na cópia de trabalho.

Para cada item atualizado, o comando exibe uma linha que inicia com um caractere informando a ação tomada. Estes caracteres têm o seguinte significado:

A
: Adicionado

D
: Removido (deleted)

U
: Atualizado (updated)

C
: Em conflito

G
: Mesclado (merged)

Um caractere na primeira coluna significa uma atualização no arquivo atual, que atualizou as propriedades do arquivo mostradas na segunda coluna.

Nomes Alternativos

up

Altera

Cópia de trabalho

Acessa o Repositório

Sim

Opções

```
--revision (-r) REV
--non-recursive (-N)
--quiet (-q)
--no-ignore
--incremental
--diff3-cmd CMD
--username USER
--password PASS
```

```
--no-auth-cache
--non-interactive
--config-dir DIR
--ignore-externals
```

Exemplos

Obtém as alterações do repositório que ocorreram desde a última operação de update:

```
$ svn update
A  newdir/toggle.c
A  newdir/disclose.c
A  newdir/launch.c
D  newdir/README
Updated to revision 32.
```

Você também pode “atualizar” sua cópia de trabalho para uma revisão mais antiga (o Subversion não tem o conceito de arquivos “aderentes” (*sticky*) que o CVS tem; consulte Apêndice B, *Subversion para Usuários de CVS*):

```
$ svn update -r30
A  newdir/README
D  newdir/toggle.c
D  newdir/disclose.c
D  newdir/launch.c
U  foo.c
Updated to revision 30.
```

Se você quiser examinar uma revisão mais antiga de um único arquivo, você pode querer usar **svn cat** no lugar—ele não altera sua cópia de trabalho.

svnadmin

O **svnadmin** é a ferramenta administrativa para monitoramento e manutenção de seu repositório Subversion. Para informações detalhadas, veja “svnadmin”.

Como o **svnadmin** trabalha com acesso direto ao repositório (e assim só pode ser usado na própria máquina onde se encontra o repositório), ele se refere ao repositório com um caminho e não com uma URL.

Opções do svnadmin

`--bdb-log-keep`
: (Berkeley DB specific) Disable automatic log removal of database log files. Having these log files around can be convenient if you need to restore from a catastrophic repository failure.

`--bdb-txn-nosync`
: (Berkeley DB specific) Disables fsync when committing database transactions. Used with the **svnadmin create** command to create a Berkeley DB backed repository with `DB_TXN_NOSYNC` enabled (which improves speed but has some risks associated with it).

`--bypass-hooks`
: Bypass the repository hook system.

`--clean-logs`
: Removes unused Berkeley DB logs.

`--force-uuid`
: By default, when loading data into repository that already contains revisions, **svnadmin** will ignore the `UUID` from the dump stream. This option will cause the repository's `UUID` to be set to the `UUID` from the stream.

`--ignore-uuid`
: By default, when loading an empty repository, **svnadmin** will ignore the `UUID` from the dump stream. This option will force that UUID to be ignored (useful for overriding your configuration file if it has `--force-uuid` set).

`--incremental`
: Dump a revision only as a diff against the previous revision, instead of the usual fulltext.

`--parent-dir` *`DIR`*
: When loading a dump file, root paths at *`DIR`* instead of `/`.

`--revision` (`-r`) *`ARG`*
: Specify a particular revision to operate on.

`--quiet`
: Do not show normal progress—show only errors.

`--use-post-commit-hook`
: When loading a dump file, run the repository's post-commit hook after finalizing each newly loaded revision.

`--use-pre-commit-hook`
: When loading a dump file, run the repository's pre-commit hook before finalizing each newly loaded revision. If the hook fails, abort the commit and terminate the load process.

svnadmin Subcommands

Nome

svnadmin create — Create a new, empty repository.

Sinopse

```
svnadmin create REPOS_PATH
```

Descrição

Create a new, empty repository at the path provided. If the provided directory does not exist, it will be created for you.[1] As of Subversion 1.2, **svnadmin** creates new repositories with the `fsfs` filesystem backend by default.

Opções

```
--bdb-txn-nosync
--bdb-log-keep
--config-dir DIR
--fs-type TYPE
```

Exemplos

Creating a new repository is just this easy:

```
$ svnadmin create /usr/local/svn/repos
```

In Subversion 1.0, a Berkeley DB repository is always created. In Subversion 1.1, a Berkeley DB repository is the default repository type, but an FSFS repository can be created using the `--fs-type` option:

```
$ svnadmin create /usr/local/svn/repos --fs-type fsfs
```

[1]Remember, **svnadmin** works only with local *paths*, not *URLs*.

Nome

svnadmin deltify — Deltify changed paths in a revision range.

Sinopse

```
svnadmin deltify [-r LOWER[:UPPER]] REPOS_PATH
```

Descrição

svnadmin deltify exists in current versions of Subversion only for historical reasons. This command is deprecated and no longer needed.

It dates from a time when Subversion offered administrators greater control over compression strategies in the repository. This turned out to be a lot of complexity for *very* little gain, and this “feature” was deprecated.

Opções

```
--revision (-r) REV
--quiet (-q)
```

Nome

svnadmin dump — Dump the contents of filesystem to stdout.

Sinopse

```
svnadmin dump REPOS_PATH [-r LOWER[:UPPER]] [--incremental]
```

Descrição

Dump the contents of filesystem to stdout in a “dumpfile” portable format, sending feedback to stderr. Dump revisions *`LOWER`* rev through *`UPPER`* rev. If no revisions are given, dump all revision trees. If only *`LOWER`* is given, dump that one revision tree. See “Migrating Repository Data Elsewhere” for a practical use.

By default, the Subversion dumpfile stream contains a single revision (the first revision in the requested revision range) in which every file and directory in the repository in that revision is presented as if that whole tree was added at once, followed by other revisions (the remainder of the revisions in the requested range) which contain only the files and directories which were modified in those revisions. For a modified file, the complete fulltext representation of its contents, as well as all of its properties, are presented in the dumpfile; for a directory, all of its properties are presented.

There are two useful options which modify the dumpfile generator's behavior. The first is the `--incremental` option, which simply causes that first revision in the dumpfile stream to contain only the files and directories modified in that revision, instead of being presented as the addition of a new tree, and in exactly the same way that every other revision in the dumpfile is presented. This is useful for generating a relatively small dumpfile to be loaded into another repository which already has the files and directories that exist in the original repository.

The second useful option is `--deltas`. This option causes **svnadmin dump** to, instead of emitting fulltext representations of file contents and property lists, emit only deltas of those items against their previous versions. This reduces (in some cases, drastically) the size of the dumpfile that **svnadmin dump** creates. There are, however, disadvantages to using this option—deltified dumpfiles are more CPU intensive to create, cannot be operated on by **svndumpfilter**, and tend not to compress as well as their non-deltified counterparts when using third-party tools like **gzip** and **bzip2**.

Opções

```
--revision (-r) REV
--incremental
--quiet (-q)
--deltas
```

Exemplos

Dump your whole repository:

```
$ svnadmin dump /usr/local/svn/repos
SVN-fs-dump-format-version: 1
Revision-number: 0
* Dumped revision 0.
Prop-content-length: 56
Content-length: 56
…
```

Incrementally dump a single transaction from your repository:

```
$ svnadmin dump /usr/local/svn/repos -r 21 --incremental
* Dumped revision 21.
SVN-fs-dump-format-version: 1
Revision-number: 21
Prop-content-length: 101
Content-length: 101
…
```

Nome

svnadmin help — Help!

Sinopse

```
svnadmin help [SUBCOMMAND...]
```

Descrição

This subcommand is useful when you're trapped on a desert island with neither a net connection nor a copy of this book.

Nomes Alternativos

?, h

Nome

svnadmin hotcopy — Make a hot copy of a repository.

Sinopse

```
svnadmin hotcopy REPOS_PATH NEW_REPOS_PATH
```

Descrição

This subcommand makes a full “hot” backup of your repository, including all hooks, configuration files, and, of course, database files. If you pass the `--clean-logs` option, **svnadmin** will perform a hotcopy of your repository, and then remove unused Berkeley DB logs from the original repository. You can run this command at any time and make a safe copy of the repository, regardless of whether other processes are using the repository.

Opções

```
--clean-logs
```

As described in “Berkeley DB”, hot-copied Berkeley DB repositories are *not* portable across operating systems, nor will they work on machines with a different “endianness” than the machine where they were created.

Nome

svnadmin list-dblogs — Ask Berkeley DB which log files exist for a given Subversion repository (applies only to repositories using the `bdb` backend).

Sinopse

```
svnadmin list-dblogs REPOS_PATH
```

Descrição

Berkeley DB creates logs of all changes to the repository, which allow it to recover in the face of catastrophe. Unless you enable `DB_LOG_AUTOREMOVE`, the log files accumulate, although most are no longer used and can be deleted to reclaim disk space. See “Managing Disk Space” for more information.

Nome

svnadmin list-unused-dblogs — Ask Berkeley DB which log files can be safely deleted (applies only to repositories using the bdb backend).

Sinopse

```
svnadmin list-unused-dblogs REPOS_PATH
```

Descrição

Berkeley DB creates logs of all changes to the repository, which allow it to recover in the face of catastrophe. Unless you enable DB_LOG_AUTOREMOVE, the log files accumulate, although most are no longer used and can be deleted to reclaim disk space. See “Managing Disk Space” for more information.

Exemplos

Remove all unused log files from a repository:

```
$ svnadmin list-unused-dblogs /path/to/repos
/path/to/repos/log.0000000031
/path/to/repos/log.0000000032
/path/to/repos/log.0000000033

$ svnadmin list-unused-dblogs /path/to/repos | xargs rm
## disk space reclaimed!
```

Nome

svnadmin load — Read a “dumpfile”-formatted stream from stdin.

Sinopse

```
svnadmin load REPOS_PATH
```

Descrição

Read a “dumpfile”-formatted stream from stdin, committing new revisions into the repository's filesystem. Send progress feedback to stdout.

Opções

```
--quiet (-q)
--ignore-uuid
--force-uuid
--use-pre-commit-hook
--use-post-commit-hook
--parent-dir
```

Example

This shows the beginning of loading a repository from a backup file (made, of course, with **svnadmin dump**):

```
$ svnadmin load /usr/local/svn/restored < repos-backup
<<< Started new txn, based on original revision 1
     * adding path : test ... done.
     * adding path : test/a ... done.
…
```

Or if you want to load into a subdirectory:

```
$ svnadmin load --parent-dir new/subdir/for/project /usr/local/svn/restored < repos-bac
<<< Started new txn, based on original revision 1
     * adding path : test ... done.
     * adding path : test/a ... done.
…
```

Nome

svnadmin lslocks — Print descriptions of all locks.

Sinopse

```
svnadmin lslocks REPOS_PATH
```

Descrição

Print descriptions of all locks in a repository.

Opções

Nenhum

Example

This lists the one locked file in the repository at `/svn/repos`:

```
$ svnadmin lslocks /svn/repos
Path: /tree.jpg
UUID Token: opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753
Owner: harry
Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)
Expires:
Comment (1 line):
Rework the uppermost branches on the bald cypress in the foreground.
```

Nome

svnadmin lstxns — Print the names of all uncommitted transactions.

Sinopse

```
svnadmin lstxns REPOS_PATH
```

Descrição

Print the names of all uncommitted transactions. See “Removing dead transactions” for information on how uncommitted transactions are created and what you should do with them.

Exemplos

List all outstanding transactions in a repository.

```
$ svnadmin lstxns /usr/local/svn/repos/
1w
1x
```

Nome

svnadmin recover — Bring a repository database back into a consistent state (applies only to repositories using the `bdb` backend). In addition, if `repos/conf/passwd` does not exist, it will create a default password file .

Sinopse

```
svnadmin recover REPOS_PATH
```

Descrição

Run this command if you get an error indicating that your repository needs to be recovered.

Opções

```
--wait
```

Exemplos

Recover a hung repository:

```
$ svnadmin recover /usr/local/svn/repos/
Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 34.
```

Recovering the database requires an exclusive lock on the repository. (This is a “database lock”; see Os três significados de trava.) If another process is accessing the repository, then **svnadmin recover** will error:

```
$ svnadmin recover /usr/local/svn/repos
svn: Failed to get exclusive repository access; perhaps another process
such as httpd, svnserve or svn has it open?

$
```

The `--wait` option, however, will cause **svnadmin recover** to wait indefinitely for other processes to disconnect:

```
$ svnadmin recover /usr/local/svn/repos --wait
Waiting on repository lock; perhaps another process has it open?

### time goes by…

Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 34.
```

Nome

svnadmin rmlocks — Unconditionally remove one or more locks from a repository.

Sinopse

```
svnadmin rmlocks REPOS_PATH LOCKED_PATH...
```

Descrição

Remove lock from each *LOCKED_PATH*.

Opções

Nenhum

Example

This deletes the locks on `tree.jpg` and `house.jpg` in the repository at `/svn/repos`

```
$ svnadmin rmlocks /svn/repos tree.jpg house.jpg
Removed lock on '/tree.jpg.
Removed lock on '/house.jpg.
```

Nome

svnadmin rmtxns — Delete transactions from a repository.

Sinopse

```
svnadmin rmtxns REPOS_PATH TXN_NAME...
```

Descrição

Delete outstanding transactions from a repository. This is covered in detail in “Removing dead transactions”.

Opções

```
--quiet (-q)
```

Exemplos

Remove named transactions:

```
$ svnadmin rmtxns /usr/local/svn/repos/ 1w 1x
```

Fortunately, the output of **lstxns** works great as the input for **rmtxns**:

```
$ svnadmin rmtxns /usr/local/svn/repos/  `svnadmin lstxns /usr/local/svn/repos/`
```

Which will remove all uncommitted transactions from your repository.

Nome

svnadmin setlog — Set the log-message on a revision.

Sinopse

```
svnadmin setlog REPOS_PATH -r REVISION FILE
```

Descrição

Set the log-message on revision REVISION to the contents of FILE.

This is similar to using **svn propset --revprop** to set the `svn:log` property on a revision, except that you can also use the option `--bypass-hooks` to avoid running any pre- or post-commit hooks, which is useful if the modification of revision properties has not been enabled in the pre-revprop-change hook.

Revision properties are not under version control, so this command will permanently overwrite the previous log message.

Opções

```
--revision (-r) REV
--bypass-hooks
```

Exemplos

Set the log message for revision 19 to the contents of the file `msg`:

```
$ svnadmin setlog /usr/local/svn/repos/ -r 19 msg
```

Nome

svnadmin verify — Verify the data stored in the repository.

Sinopse

```
svnadmin verify REPOS_PATH
```

Descrição

Run this command if you wish to verify the integrity of your repository. This basically iterates through all revisions in the repository by internally dumping all revisions and discarding the output—it's a good idea to run this on a regular basis to guard against latent hard disk failures and “bitrot”. If this command fails—which it will do at the first sign of a problem—that means that your repository has at least one corrupted revision and you should restore the corrupted revision from a backup (you did make a backup, didn't you?).

Exemplos

Verify a hung repository:

```
$ svnadmin verify /usr/local/svn/repos/
* Verified revision 1729.
```

svnlook

svnlook é um comando de console útil para examinar diferentes aspectos do repositório Subversion. Ele não faz nenhuma mudança no repositório—é usado mesmo para dar uma dar uma “espiada”. **svnlook** é usando tipicamente usado pelos hooks do repositório, mas o administrador do repositório pode achá-lo útil como ferramente de diagnóstico.

Já que **svnlook** funciona via acesso direto ao repositório, (e por isso só pode ser usado em máquinas que tenham repositórios), ele se refere ao repositório por um caminho, não uma URL.

Se nenhuma revisão ou transação for especificada, o padrão do **svnlook** é da revisão mais jovem(mais recente) do repositório.

Opções do svnlook

Opções no **svnlook** são globais, assim como no **svn** e **svnadmin**; entretanto, a maioria das opções apenas se aplicam a um commando já que as funcionalidades do **svnlook** é (intencionalmente) limitado ao escopo.

`--no-diff-deleted`
: Previne o **svnlook** de mostrar as diferenças entre arquivos deletados. O comportamento padrão quando um arquivo é deletado numa transação/revisão é mostrar as mesmas diferenças que você veria se tivesse deixado o arquivo mas apagado seu conteudo.

`--revision` (`-r`)
: Especifica uma revisão em particular que você deseja examinar.

`--revprop`
: Opera uma propriedade da revisão ao invés da propriedade especificada para o arquivo ou diretório. Essa opção exige que você passe a revisão com a opção `--revision` (`-r`).

`--transaction` (`-t`)
: Especifica um ID de transação particular que você deseja examinar.

`--show-ids`
: Mostrar o ID da revisão do nodo do sistema de arquivos para cada caminho da árvore do sistema de arquivos.

Sub-comandos do svnlook

Nome

autor svnlook — Mostrar o autor.

Sinopse

```
svnlook author REPOS_PATH
```

Descrição

Mostrar o autor da revisão ou transação no repositório.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

svnlook author é útil, mas não muito excitante:

```
$ svnlook author -r 40 /usr/local/svn/repos
sally
```

Nome

svnlook cat — Mostra o conteudo de um arquivo.

Sinopse

```
svnlook cat REPOS_PATH PATH_IN_REPOS
```

Descrição

Mostra o conteudo de um arquivo.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

Isto motra o conteudo de um arquivo em uma transação `ax8`, localizado no `/trunk/README`:

```
$ svnlook cat -t ax8 /usr/local/svn/repos /trunk/README

               Subversion, a version control system.
               =====================================

$LastChangedDate: 2003-07-17 10:45:25 -0500 (Thu, 17 Jul 2003) $

Contents:

     I. A FEW POINTERS
    II. DOCUMENTATION
   III. PARTICIPATING IN THE SUBVERSION COMMUNITY
…
```

Nome

svnlook changed — Mostra os caminhos que foram mudados.

Sinopse

```
svnlook changed REPOS_PATH
```

Descrição

Mostra os caminhos que foram mudados em uma revisão ou transação particular, assim como “svn update-style” as letras de estatus nas duas primeiras colunas:

'A '
: Item adicionado ao repositório.

'D '
: Item apagado do respostório.

'U '
: Conteúdo do arquivo foi mudado.

' U'
: Propriedades do tem mudados.--FIXME Note the leading space.--

'UU'
: Conteudo e propriedades mudados.

Arquivos e diretórios podem ser distinguidos, como os caminhos dos diretórios são mostrados com caracter '/'.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

Isto mostra a lista de todos os diretórios e arquivos mudados na revisão 39 em um diretório de teste. Note que a primeira mudança é um diretório, como evidenciado pela /:

```
$ svnlook changed -r 39 /usr/local/svn/repos
A   trunk/vendors/deli/
A   trunk/vendors/deli/chips.txt
A   trunk/vendors/deli/sandwich.txt
A   trunk/vendors/deli/pickle.txt
U   trunk/vendors/baker/bagel.txt
 U  trunk/vendors/baker/croissant.txt
UU  trunk/vendors/baker/pretzel.txt
D   trunk/vendors/baker/baguette.txt
```

Nome

svnlook date — Mostrar data-hora.

Sinopse

```
svnlook date REPOS_PATH
```

Descrição

Mostrar data-hora de uma revisão ou transação em um repositório.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

Mostra a data da revisão 40 de um repositório de teste:

```
$ svnlook date -r 40 /tmp/repos/
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
```

Nome

svnlook diff — Mostra diferenças de arquivos e propriedades que foram mudados.

Sinopse

```
svnlook diff REPOS_PATH
```

Descrição

Mostra no estilo GNU diferençasde arquivos e propriedades que foram mudados.

Opções

```
--revision (-r) REV
--transaction (-t)
--no-diff-added
--no-diff-deleted
```

Exemplos

Isto mostra um novo arquivo adicionado, deletado e copiado:

```
$ svnlook diff -r 40 /usr/local/svn/repos/
Copied: egg.txt (from rev 39, trunk/vendors/deli/pickle.txt)

Added: trunk/vendors/deli/soda.txt
==============================================================================

Modified: trunk/vendors/deli/sandwich.txt
==============================================================================
--- trunk/vendors/deli/sandwich.txt (original)
+++ trunk/vendors/deli/sandwich.txt 2003-02-22 17:45:04.000000000 -0600
@@ -0,0 +1 @@
+Don't forget the mayo!

Modified: trunk/vendors/deli/logo.jpg
==============================================================================
(Binary files differ)

Deleted: trunk/vendors/deli/chips.txt
==============================================================================

Deleted: trunk/vendors/deli/pickle.txt
==============================================================================
```

Se um arquivo tem um conteudo que não é texto, propriedade `svn:mime-type`, então as diferenças não são explicitamente mostradas.

Nome

svnlook dirs-changed — Mostra os diretórios que foram mudados.

Sinopse

```
svnlook dirs-changed REPOS_PATH
```

Descrição

Mostra os diretórios que foram mudados (edição de propriedade) ou tiveram seus filhos mudados.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

Isto mostra os difertórios que foram mudados na revisão 40 no nosso respostório de exemplo:

```
$ svnlook dirs-changed -r 40 /usr/local/svn/repos
trunk/vendors/deli/
```

Nome

svnlook help — Help!

Sinopses

`Também svnlook -h e svnlook -?.`

Descrição

Mostra a mensagem de ajuda para o svnlook. Este comando, como seu irmão **svn help**, é também seu amigo, mesmo que você não ligue mais pra ele e tenha esquecido de convida-lo para sua última festa.

Nomes alternativos

?, h

Nome

svnlook history — Mostra informções sobre o histórico de um caminhos em um repositório (ou da raiz do diretório se nenhum caminho for informado).

Sinopse

```
svnlook history REPOS_PATH [PATH_IN_REPOS]
```

Descrição

Mostra informações sobre histórico de um caminho em um repositório (ou da raiz do diretório se nenhum caminho for informado).

Opções

```
--revision (-r) REV
--show-ids
```

Exemplos

Isto mostra o histórico de um caminho `/tags/1.0` da revisão 20 no nosso repositório de exemplo.

```
$ svnlook history -r 20 /usr/local/svn/repos /tags/1.0 --show-ids
REVISION   PATH <ID>
--------   ---------
      19   /tags/1.0 <1.2.12>
      17   /branches/1.0-rc2 <1.1.10>
      16   /branches/1.0-rc2 <1.1.x>
      14   /trunk <1.0.q>
      13   /trunk <1.0.o>
      11   /trunk <1.0.k>
       9   /trunk <1.0.g>
       8   /trunk <1.0.e>
       7   /trunk <1.0.b>
       6   /trunk <1.0.9>
       5   /trunk <1.0.7>
       4   /trunk <1.0.6>
       2   /trunk <1.0.3>
       1   /trunk <1.0.2>
```

Nome

svnlook info — Mostra o autor, data-hora, tamanho da mensagem de log, e a mensagem de log.

Sinopse

```
svnlook info REPOS_PATH
```

Descrição

Mostra o autor, data-hora, tamanho da mensagem de log, e a mensagem de log.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

Isto mostra a saída para a revisão 40 no nosso repositório de exemplo.

```
$ svnlook info -r 40 /usr/local/svn/repos
sally
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
15
Rearrange lunch.
```

Nome

svnlook lock — Se o lock existir no caminho do repositório, o descreve.

Sinopse

```
svnlook lock REPOS_PATH PATH_IN_REPOS
```

Descrição

Mostra todas as informações disponíveis para o lock no *`PATH_IN_REPOS`*. Se *`PATH_IN_REPOS`* não estiver lockado, não mostra nada.

Opções

Nada

Exemplos

Descreve o lock do arquivo `tree.jpg`.

```
$ svnlook lock /svn/repos tree.jpg
UUID Token: opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753
Owner: harry
Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)
Expires:
Comment (1 line):
Rework the uppermost branches on the bald cypress in the foreground.
```

Nome

svnlook log — Mostra a mensagem de log.

Sinopse

```
svnlook log REPOS_PATH
```

Descrição

Mostra a mensagem de log.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

Isto mostra o log de saída para a revisão 40 no nosso repositório de exemplo:

```
$ svnlook log /tmp/repos/
Rearrange lunch.
```

Nome

svnlook propget — Print the raw value of a property on a path in the repository.

Sinopse

```
svnlook propget REPOS_PATH PROPNAME [PATH_IN_REPOS]
```

Descrição

List the value of a property on a path in the repository.

Nomes Alternativos

pg, pget

Opções

```
--revision (-r) REV
--transaction (-t)
--revprop
```

Exemplos

This shows the value of the “seasonings” property on the file `/trunk/sandwich` in the `HEAD` revision:

```
$ svnlook pg /usr/local/svn/repos seasonings /trunk/sandwich
mustard
```

Nome

svnlook proplist — Print the names and values of versioned file and directory properties.

Sinopse

```
svnlook proplist REPOS_PATH [PATH_IN_REPOS]
```

Descrição

List the properties of a path in the repository. With `--verbose`, show the property values too.

Nomes Alternativos

pl, plist

Opções

```
--revision (-r) REV
--transaction (-t)
--verbose (-v)
--revprop
```

Exemplos

This shows the names of properties set on the file `/trunk/README` in the `HEAD` revision:

```
$ svnlook proplist /usr/local/svn/repos /trunk/README
  original-author
  svn:mime-type
```

This is the same command as in the previous example, but this time showing the property values as well:

```
$ svnlook --verbose proplist /usr/local/svn/repos /trunk/README
  original-author : fitz
  svn:mime-type : text/plain
```

Nome

svnlook tree — Print the tree.

Sinopse

```
svnlook tree REPOS_PATH [PATH_IN_REPOS]
```

Descrição

Print the tree, starting at *PATH_IN_REPOS* (if supplied, at the root of the tree otherwise), optionally showing node revision IDs.

Opções

```
--revision (-r) REV
--transaction (-t)
--show-ids
```

Exemplos

This shows the tree output (with node-IDs) for revision 40 in our sample repository:

```
$ svnlook tree -r 40 /usr/local/svn/repos --show-ids
/ <0.0.2j>
 trunk/ <p.0.2j>
  vendors/ <q.0.2j>
   deli/ <1g.0.2j>
    egg.txt <1i.e.2j>
    soda.txt <1k.0.2j>
    sandwich.txt <1j.0.2j>
```

Nome

svnlook uuid — Print the repository's `UUID`.

Sinopse

```
svnlook uuid REPOS_PATH
```

Descrição

Print the `UUID` for the repository. the UUID is the repository's *u*niversal *u*nique *id*entifier. The Subversion client uses this identifier to differentiate between one repository and another.

Exemplos

```
$ svnlook uuid /usr/local/svn/repos
e7fe1b91-8cd5-0310-98dd-2f12e793c5e8
```

Nome

svnlook youngest — Print the youngest revision number.

Sinopse

```
svnlook youngest REPOS_PATH
```

Descrição

Print the youngest revision number of a repository.

Exemplos

This shows the youngest revision of our sample repository:

```
$ svnlook youngest /tmp/repos/
42
```

svnsync

svnsync is the Subversion remote repository mirroring tool. Put simply, it allows you to replay the revisions of one repository into another one.

In any mirroring scenario, there are two repositories: the source repository, and the mirror (or “sink”) repository. The source repository is the repository from which **svnsync** pulls revisions. The mirror repository is the destination for the revisions pulled from the source repository. Each of the repositories may be local or remote—they are only ever addressed by their URLs.

The **svnsync** process requires only read access to the source repository; it never attempts to modify it. But obviously, **svnsync** requires both read and write access to the mirror repository.

svnsync is very sensitive to changes made in the mirror repository that weren't made as part of a mirroring operation. To prevent this from happening, it's best if the **svnsync** process is the only process permitted to modify the mirror repository.

svnsync Options

`--config-dir` *`DIR`*
: Instructs Subversion to read configuration information from the specified directory instead of the default location (`.subversion` in the user's home directory).

`--no-auth-cache`
: Prevents caching of authentication information (e.g. username and password) in the Subversion administrative directories.

`--non-interactive`
: In the case of an authentication failure, or insufficient credentials, prevents prompting for credentials (e.g. username or password). This is useful if you're running Subversion inside of an automated script and it's more appropriate to have Subversion fail than to prompt for more information.

`--password` *`PASS`*
: Indicates that you are providing your password for authentication on the command line—otherwise, if it is needed, Subversion will prompt you for it.

`--username` *`NAME`*
: Indicates that you are providing your username for authentication on the command line—otherwise, if it is needed, Subversion will prompt you for it.

svnsync Subcommands

Here are the various subcommands:

Nome

svnsync copy-revprops — Copy all revision properties for a given revision from the source repository to the mirror repository.

Sinopse

```
svnsync copy-revprops DEST_URL REV
```

Descrição

Because Subversion revision properties can be changed at any time, it's possible that the properties for some revision might be changed after that revision has already been synchronized to another repository. Because the **svnsync synchronize** command operates only on the range of revisions that have not yet been synchronized, it won't notice a revision property change outside that range. Left as is, this causes a deviation in the values of that revision's properties between the source and mirror repositories. **svnsync copy-revprops** is the answer to this problem. Use it to re-synchronize the revision properties for a particular revision.

Nomes Alternativos

Nenhum

Opções

```
--non-interactive
--no-auth-cache
--username NAME
--password PASS
--config-dir DIR
```

Exemplos

Re-synchronize revision properties for a single revision:

```
$ svnsync copy-revprops file:///opt/svn/repos-mirror 6
Copied properties for revision 6.
$
```

Nome

svnsync initialize — Initialize a destination repository for synchronization from another repository.

Sinopse

```
svnsync initialize DEST_URL SOURCE_URL
```

Descrição

svnsync initialize verifies that a repository meets the requirements of a new mirror repository—that it has no previous existing version history, and that it allows revision property modifications—and records the initial administrative information which associates the mirror repository with the source repository. This is the first **svnsync** operation you run on a would-be mirror repository.

Nomes Alternativos

init

Opções

```
--non-interactive
--no-auth-cache
--username NAME
--password PASS
--config-dir DIR
```

Exemplos

Fail to initialize a mirror repository due to inability to modify revision properties:

```
$ svnsync initialize file:///opt/svn/repos-mirror http://svn.example.com/repos
svnsync: Repository has not been enabled to accept revision propchanges;
ask the administrator to create a pre-revprop-change hook
$
```

Initialize a repository as a mirror, having already created a pre-revprop-change hook which permits all revision property changes:

```
$ svnsync initialize file:///opt/svn/repos-mirror http://svn.example.com/repos
Copied properties for revision 0.
$
```

Nome

svnsync synchronize — Transfer all pending revisions from the source repository to the mirror repository.

Sinopse

```
svnsync synchronize DEST_URL
```

Descrição

The **svnsync synchronize** command does all the heavy lifting of a repository mirroring operation. After consulting with the mirror repository to see which revisions have already been copied into it, it then begins copying any not-yet-mirrored revisions from the source repository.

svnsync synchronize can be gracefully cancelled and restarted.

Nomes Alternativos

sync

Opções

```
--non-interactive
--no-auth-cache
--username NAME
--password PASS
--config-dir DIR
```

Exemplos

Copy unsynchronized revisions from the source repository to the mirror repository:

```
$ svnsync synchronize file:///opt/svn/repos-mirror
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
Copied properties for revision 3.
…
Committed revision 45.
Copied properties for revision 45.
Committed revision 46.
Copied properties for revision 46.
Committed revision 47.
Copied properties for revision 47.
$
```

svnserve

svnserve allows access to Subversion repositories using Subversion's custom network protocol.

You can run **svnserve** as a standalone server process (for clients that are using the `svn://` access method); you can have a daemon such as **inetd** or **xinetd** launch it for you on demand (also for `svn://`), or you can have **sshd** launch it on demand for the `svn+ssh://` access method.

Regardless of the access method, once the client has selected a repository by transmitting its URL, **svnserve** reads a file named `conf/svnserve.conf` in the repository directory to determine repository-specific settings such as what authentication database to use and what authorization policies to apply. See “svnserve, um servidor especializado” for details of the `svnserve.conf` file.

svnserve Options

Unlike the previous commands we've described, **svnserve** has no subcommands—**svnserve** is controlled exclusively by options.

`--daemon` (`-d`)
: Causes **svnserve** to run in daemon mode. **svnserve** backgrounds itself and accepts and serves TCP/IP connections on the svn port (3690, by default).

`--listen-port=`*`PORT`*
: Causes svnserve to listen on *`PORT`* when run in daemon mode. (FreeBSD daemons only listen on tcp6 by default—this option tells them to also listen on tcp4.)

`--listen-host=`*`HOST`*
: Causes **svnserve** to listen on the interface specified by *`HOST`*, which may be either a hostname or an IP address.

`--foreground`
: When used together with `-d`, this option causes **svnserve** to stay in the foreground. This option is mainly useful for debugging.

`--inetd` (`-i`)
: Causes **svnserve** to use the stdin/stdout file descriptors, as is appropriate for a daemon running out of **inetd**.

`--help` (`-h`)
: Displays a usage summary and exits.

`--version`
: Displays version information, a list of repository back-end modules available, and exits.

`--root=`*`ROOT`* (`-r=`*`ROOT`*)
: Sets the virtual root for repositories served by **svnserve**. The pathname in URLs provided by the client will be interpreted relative to this root, and will not be allowed to escape this root.

`--tunnel` (`-t`)
: Causes **svnserve** to run in tunnel mode, which is just like the **inetd** mode of operation (both modes serve one connection over stdin/stdout, then exit), except that the connection is considered to be pre-authenticated with the username of the current uid. This flag is automatically passed for you by the client when running over a tunnel agent such as **ssh**. That means there's rarely any need for *you* to pass this option to **svnserve**. So if you find yourself typing `svnserve --tunnel` on the command line, and wondering what to do next, see “Tunelamento sobre SSH”.

`--tunnel-user NAME`
: Used in conjunction with the `--tunnel` option; tells svnserve to assume that *`NAME`* is the authenticated user, rather than the UID of the svnserve process. Useful for users wishing to share a single system account over SSH, but maintaining separate commit identities.

`--threads` (`-T`)
: When running in daemon mode, causes **svnserve** to spawn a thread instead of a process for each connection (e.g. for when running on Windows). The **svnserve** process still backgrounds itself at startup time.

`--listen-once` (`-X`)
: Causes **svnserve** to accept one connection on the svn port, serve it, and exit. This option is mainly useful for debugging.

svnversion

Nome

svnversion — Summarize the local revision(s) of a working copy.

Sinopse

```
svnversion [OPTIONS] [WC_PATH [TRAIL_URL]]
```

Descrição

svnversion is a program for summarizing the revision mixture of a working copy. The resultant revision number, or revision range, is written to standard output.

It's common to use this output in your build process when defining the version number of your program.

`TRAIL_URL`, if present, is the trailing portion of the URL used to determine if *`WC_PATH`* itself is switched (detection of switches within *`WC_PATH`* does not rely on *`TRAIL_URL`*).

When *`WC_PATH`* is not defined, the current directory will be used as the working copy path. *`TRAIL_URL`* cannot be defined if *`WC_PATH`* is not explicitly given.

Opções

Like **svnserve**, **svnversion** has no subcommands, it only has options.

`--no-newline` (`-n`)
: Omit the usual trailing newline from the output.

`--committed` (`-c`)
: Use the last-changed revisions rather than the current (i.e., highest locally available) revisions.

`--help` (`-h`)
: Print a help summary.

`--version`
: Print the version of **svnversion** and exit with no error.

Exemplos

If the working copy is all at the same revision (for example, immediately after an update), then that revision is printed out:

```
$ svnversion
4168
```

You can add *`TRAIL_URL`* to make sure that the working copy is not switched from what you expect. Note that the *`WC_PATH`* is required in this command:

```
$ svnversion . /repos/svn/trunk
4168
```

For a mixed-revision working copy, the range of revisions present is printed:

```
$ svnversion
4123:4168
```

If the working copy contains modifications, a trailing "M" is added:

```
$ svnversion
4168M
```

If the working copy is switched, a trailing "S" is added:

```
$ svnversion
4168S
```

Thus, here is a mixed-revision, switched working copy containing some local modifications:

```
$ svnversion
4212:4168MS
```

If invoked on a directory that is not a working copy, **svnversion** assumes it is an exported working copy and prints "exported":

```
$ svnversion
exported
```

mod_dav_svn

Nome

`mod_dav_svn` Configuration Directives — Apache configuration directives for serving Subversion repositories through Apache HTTP Server.

Descrição

This section briefly describes each of the Subversion Apache configuration directives. For an in-depth description of configuring Apache with Subversion, see “httpd, o servidor HTTP Apache”.)

Directives

`DAV svn`
: This directive must be included in any `Directory` or `Location` block for a Subversion repository. It tells httpd to use the Subversion backend for mod_dav to handle all requests.

`SVNAutoversioning On`
: This directive allows write requests from WebDAV clients to result in automatic commits. A generic log message is auto-generated and attached to each revision. If you enable Autoversioning, you'll likely want to set `ModMimeUsePathInfo On` so that `mod_mime` can set `svn:mime-type` to the correct mime-type automatically (as best as `mod_mime` is able to, of course). For more information, see Apêndice C, *WebDAV e Autoversionamento*

`SVNPath`
: This directive specifies the location in the filesystem for a Subversion repository's files. In a configuration block for a Subversion repository, either this directive or `SVNParentPath` must be present, but not both.

`SVNSpecialURI`
: Specifies the URI component (namespace) for special Subversion resources. The default is “`!svn`”, and most administrators will never use this directive. Only set this if there is a pressing need to have a file named `!svn` in your repository. If you change this on a server already in use, it will break all of the outstanding working copies and your users will hunt you down with pitchforks and flaming torches.

`SVNReposName`
: Specifies the name of a Subversion repository for use in `HTTP GET` responses. This value will be prepended to the title of all directory listings (which are served when you navigate to a Subversion repository with a web browser). This directive is optional.

`SVNIndexXSLT`
: Specifies the URI of an XSL transformation for directory indexes. This directive is optional.

`SVNParentPath`
: Specifies the location in the filesystem of a parent directory whose child directories are Subversion repositories. In a configuration block for a Subversion repository, either this directive or `SVNPath` must be present, but not both.

`SVNPathAuthz`
: Control path-based authorization by enabling or disabling subrequests. See “Desabilitando Verificação baseada em Caminhos” for details.

Subversion properties

Subversion allows users to invent arbitrarily-named versioned properties on files and directories, as well as unversioned properties on revisions. The only restriction is on properties whose names begin with `svn:` (those are reserved for Subversion's own use). While these properties may be set by users to control Subversion's behavior, users may not invent new `svn:` properties.

Versioned Properties

`svn:executable`
: If present on a file, the client will make the file executable in Unix-hosted working copies. See “Executabilidade de Arquivo”.

`svn:mime-type`
: If present on a file, the value indicates the file's mime-type. This allows the client to decide whether line-based contextual merging is safe to perform during updates, and can also affect how the file behaves when fetched via web browser. See “Tipo de Conteúdo do Arquivo”.

`svn:ignore`
: If present on a directory, the value is a list of *unversioned* file patterns to be ignored by **svn status** and other subcommands. See “Ignorando Itens Não-Versionados”

`svn:keywords`
: If present on a file, the value tells the client how to expand particular keywords within the file. See “Substituição de Palavra-Chave”.

`svn:eol-style`
: If present on a file, the value tells the client how to manipulate the file's line-endings in the working copy, and in exported trees. See “Seqüência de Caracteres de Fim-de-Linha” and svn export.

`svn:externals`
: If present on a directory, the value is a multi-line list of other paths and URLs the client should check out. See “Definições Externas”.

`svn:special`
: If present on a file, indicates that the file is not an ordinary file, but a symbolic link or other special object[1].

`svn:needs-lock`
: If present on a file, tells the client to make the file read-only in the working copy, as a reminder that the file should be locked before editing begins. See “Comunicação de Travas”.

Unversioned Properties

`svn:author`
: If present, contains the authenticated username of the person who created the revision. (If not present, then the revision was committed anonymously.)

`svn:date`
: Contains the UTC time the revision was created, in ISO 8601 format. The value comes from the *server* machine's clock, not the client's.

`svn:log`
: Contains the log message describing the revision.

`svn:autoversioned`
: If present, the revision was created via the autoversioning feature. See “Autoversionamento”.

Repository Hooks

[1]As of this writing, symbolic links are indeed the only “special” objects. But there might be more in future releases of Subversion.

Nome

start-commit — Notification of the beginning of a commit.

Descrição

The start-commit hook is run before the commit transaction is even created. It is typically used to decide if the user has commit privileges at all.

If the start-commit hook program returns a non-zero exit value, the commit is stopped before the commit transaction is even created, and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path

2. authenticated username attempting the commit

Common Uses

access control

Nome

pre-commit — Notification just prior to commit completion.

Descrição

The pre-commit hook is run just before a commit transaction is promoted to a new revision. Typically, this hook is used to protect against commits that are disallowed due to content or location (for example, your site might require that all commits to a certain branch include a ticket number from the bug tracker, or that the incoming log message is non-empty).

If the pre-commit hook program returns a non-zero exit value, the commit is aborted, the commit transaction is removed, and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. commit transaction name

Common Uses

change validation and control

Nome

post-commit — Notification of a successful commit.

Descrição

The post-commit hook is run after the transaction is committed, and a new revision created. Most people use this hook to send out descriptive emails about the commit or to notify some other tool (such as an issue tracker) that a commit has happened. Some configurations also use this hook to trigger backup processes.

The output from, and exit value returned by the post-commit hook program are ignored.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. revision number created by the commit

Common Uses

commit notification, tool integration

Nome

pre-revprop-change — Notification of a revision property change attempt.

Descrição

The pre-revprop-change hook is run immediately prior to the modification of a revision property when performed outside the scope of a normal commit. Unlike the other hooks, the default state of this one is to deny the proposed action. The hook must actually exist and return a zero exit value before a revision property modification can happen.

If the pre-revprop-change hook doesn't exist, isn't executable, or returns a non-zero exit value, no change to the property will be made, and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. revision whose property is about to be modified
3. authenticated username attempting the propchange
4. name of the property changed
5. change description: `A` (added), `D` (deleted), or `M` (modified)

Additionally, Subversion passes to the hook program via standard input the proposed value of the property.

Common Uses

access control, change validation and control

Nome

post-revprop-change — Notification of a successful revision property change.

Descrição

The post-revprop-change hook is run immediately after to the modification of a revision property when performed outside the scope of a normal commit. As can be derived from the description of its counterpart, the pre-revprop-change hook, this hook will not run at all unless the pre-revprop-change hook is implemented. It is typically used to send email notification of the property change.

The output from, and exit value returned by, the post-revprop-change hook program are ignored.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. revision whose property was modified
3. authenticated username of the person making the change
4. name of the property changed
5. change description: `A` (added), `D` (deleted), or `M` (modified)

Additionally, Subversion passes to the hook program, via standard input, the previous value of the property.

Common Uses

propchange notification

Nome

pre-lock — Notification of a path lock attempt.

Descrição

The pre-lock hook runs whenever someone attempts to lock a path. It can be used to prevent locks altogether, or to create a more complex policy specifying exactly which users are allowed to lock particular paths. If the hook notices a pre-existing lock, then it can also decide whether a user is allowed to “steal” the existing lock.

If the pre-lock hook program returns a non-zero exit value, the lock action is aborted and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. versioned path which is to be locked
3. authenticated username of the person attempting the lock

Common Uses

access control

Nome

post-lock — Notification of a successful path lock.

Descrição

The post-lock hook runs after one or more paths has been locked. It is typically used to send email notification of the lock event.

The output from and exit value returned by the post-lock hook program are ignored.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path

2. authenticated username of the person who locked the paths

Additionally, the list of paths locked is passed to the hook program via standard input, one path per line.

Common Uses

lock notification

Nome

pre-unlock — Notification of a path unlock attempt.

Descrição

The pre-unlock hook runs whenever someone attempts to remove a lock on a file. It can be used to create policies that specify which users are allowed to unlock particular paths. It's particularly important for determining policies about lock breakage. If user A locks a file, is user B allowed to break the lock? What if the lock is more than a week old? These sorts of things can be decided and enforced by the hook.

If the pre-unlock hook program returns a non-zero exit value, the unlock action is aborted and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. versioned path which is to be locked
3. authenticated username of the person attempting the lock

Common Uses

access control

Nome

post-unlock — Notification of a successful path unlock.

Descrição

The post-unlock hook runs after one or more paths has been unlocked. It is typically used to send email notification of the unlock event.

The output from and exit value returned by, the post-unlock hook program are ignored.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path

2. authenticated username of the person who unlocked the paths

Additionally, the list of paths unlocked is passed to the hook program via standard input, one path per line.

Common Uses

unlock notification

Apêndice A. Guia Rápido de Introdução ao Subversion

Se você está ansioso para ter o Subversion configurado e funcionando (e se você é daqueles que gostam de aprender fazendo), este capítulo vai lhe mostrar como criar um repositório, importar código, e então obtê-lo de volta como uma cópia de trabalho. Ao longo do caminho, damos referências aos capítulos relevantes deste livro.

Se os conceitos de controle de versão ou o modelo “copiar-modificar-mesclar” usado tanto pelo CVS quanto pelo Subversion, então você deveria ler Capítulo 1, *Conceitos Fundamentais* antes de seguir em frente.

Instalando o Subversion

O Subversion é construído sobre uma camada de portabilidade chamada APR—a biblioteca Apache Portable Runtime. A biblioteca APR provê todas as interfaces de que o Subversion precisa para funcionar em diferentes sistemas operacionais: acesso a disco, acesso à rede, gerência de memória, e por aí vai. Ainda que o Subversion seja capaz de usar o Apache como um de seus programas servidores de rede, sua dependência da APR *não* significa que o Apache seja um componente requerido. APR é uma biblioteca independente e que pode ser usada por qualquer aplicação. Isso significa, entretanto, que como o Apache, os clientes Subversion e servidores executam em quaisquer sistemas operacionais em que o servidor Apache httpd execute: Windows, Linux, todos os tipos de BSD, Mac OS X, Netware, e outros.

A maneira mais fácil de obter o Subversion é fazendo o download do pacote binário construído para seu sistema operacional. O site do Subversion (http://subversion.tigris.org) quase sempre terá estes pacotes disponíveis para download, submetidos por voluntários. O site comumente contém pacotes de instaladores gráficos para os usuários de sistemas operacionais Microsoft. Se você tem um sistema operacional Unix-like, você pode usar o sistema de pacotes nativo de sua distribuição (RPMs, DEBs, a árvore de ports, etc.) para obter o Subversion.

Como alternativa, você também pode compilar o Subversion diretamente a partir do código-fonte, ainda que esta não seja sempre uma tarefa fácil. (Se você não tem experiência em compilar pacotes de software de código aberto, ao invés disso, provavelmente seria melhor que você fizesse o download da distribuição binária!) Do site do Subversion, baixe a última versão do código-fonte. Depois de descompactá-lo, siga as instruções no arquivo `INSTALL` para compilá-lo. Note que um pacote com os fontes pode não conter tudo o que você precisa para compilar um cliente de linha de comando capaz de se comunicar com um repositório remoto. Desde o Subversion 1.4 e posteriores, as bibliotecas de que o Subversion depende (apr, apr-util, e neon) são distribuídas em um pacote de fontes em separado com o sufixo `-deps`. Estas bibliotecas, no entanto, são bastante comuns hoje em dia e é possível que já estejam instaladas em seu sistema. Se não, você precisará descompactar o pacote de dependências no mesmo diretório em que descompactou os fontes principais do Subversion. Além disso, porém, é possível que você queira obter outras dependências opcionais tais como Berkeley DB e possivelmente o servidor Apache httpd. Se você quiser uma compilação completa, certifique-se de ter todos os pacotes relacionados no arquivo `INSTALL`.

Se você é uma daquelas pessoas que gosta de usar software ainda em desenvolvimento, você também pode obter o código-fonte do Subversion a partir do repositório Subversion onde ele se encontra. Obviamente, você precisa já ter um cliente Subversion em mãos para fazer isso. Mas uma vez que você o faça, você pode obter uma cópia de trabalho do repositório dos fontes do Subversion em http://svn.collab.net/repos/svn/trunk/: [1]

[1]Perceba que a URL mencionada no exemplo acima não termina com `svn`, mas com um subdiretório chamado de `trunk`. Veja nossa discussão sobre o modelo de ramificação (*branching*) e rotulagem (*branching*) para entender as razões por trás disto.

```
$ svn checkout http://svn.collab.net/repos/svn/trunk subversion
A    subversion/HACKING
A    subversion/INSTALL
A    subversion/README
A    subversion/autogen.sh
A    subversion/build.conf
…
```

O comando acima vai criar uma cópia de trabalho da última versão (ainda não distribuída) do código-fonte do Subversion em um diretório chamado `subversion` no diretório de trabalho atual. Você pode ajustar o último argumento como quiser. Independentemente de que nome você dê ao diretório de sua nova cópia de trabalho, porém, quando esta operação concluir, você agora terá o código-fonte do Subversion. É claro que você ainda vai precisar obter algumas poucas bibliotecas auxiliares (apr, apr-util, etc.)—consulte o arquivo `INSTALL` na raiz de sua cópia de trabalho para mais detalhes.

Tutorial "Alta Velocidade"

> “Por favor, retorne o encosto de sua poltrona para a posição vertical, e verifique o travamento da mesinha à sua frente. Tripulação, preparar para decolagem….”

A seguir, você confere um breve tutorial que vai lhe conduzir pela configuração e utilização básica do Subversion. Ao final, você deve ter uma compreensão básica do uso típico do Subversion.

Os exemplos usados neste apêndice assumem que você tem o **svn**, o cliente de linha de comando do Subversion, e o **svnadmin**, a ferramenta administrativa, prontas para usar em um sistema operacional Unix-like. (Este tutorial também funciona para o prompt de comando do Windows, assumindo que você faça algumas devidas adaptações.) Também assumimos que você está usando o Subversion 1.2 ou posterior (execute **svn --version** para conferir.)

O Subversion armazena todos os dados versionados em um repositório central. Para começar, crie um novo repositório:

```
$ svnadmin create /caminho/do/repositorio
$ ls /caminho/do/repositorio
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

Este comando cria um novo diretório em `/caminho/do/repositorio` que contém um repositório do Subversion. Este diretório contém (entre outras coisas) um conjunto de arquivos de base de dados. Você não verá seus arquivos versionados olhando seu conteúdo. Para mais informações sobre criação e manutenção de repositórios, veja Capítulo 5, *Administração do Repositório*.

O Subversion não tem o conceito de “projeto”. O repositório é apenas um sistema de arquivo virtual sob controle de versão, uma grande árvore que pode conter qualquer coisa que você quiser. Alguns administradores preferem armazenar apenas um projeto em um repositório, enquanto outros preferem armazenas múltiplos projetos em um repositório colocando-os em diretórios separados. Os méritos de cada abordagem são discutidos em “Planejando a Organização do Repositório”. De qualquer forma, o repositório apenas gerencia arquivos e diretórios, então é intuitivo interpretar diretórios específicos como “projetos”. Assim, quando você vir referências a projetos ao longo deste livro, tenha em mente que estamos apenas falando sobre algum dado diretório (ou conjunto de diretórios) dentro do repositório.

Neste exemplo, assumimos que que você já tem algum tipo de projeto (um conjunto de arquivos de diretórios) que você quer importar para dentro de seu repositório Subversion recém-criado. Comece organizando seus dados dentro de um único diretório chamado `meuprojeto` (ou qualquer outro nome de sua preferência). Por motivos que ficarão mais claros posteriormente (veja Capítulo 4, *Fundir e Ramificar*), a estrutura da árvore de seu projeto deve conter três diretórios internos chamados de `branches`, `tags`, e `trunk`. O diretório `trunk` deve conter todos os seus dados, enquanto que os diretórios `branches` e `tags` são vazios:

```
/tmp/meuprojeto/branches/
/tmp/meuprojeto/tags/
/tmp/meuprojeto/trunk/
                     foo.c
                     bar.c
                     Makefile
                     …
```

Os subdiretórios `branches`, `tags`, e `trunk` não são atualmente requeridos pelo subversion. São meramente uma convenção que você provavelmente também vai querer seguir daqui para a frente.

Uma vez você sua árvore de dados esteja pronta, importe os dados para dentro do repositório com o comando **svn import** (veja “Colocando dados em seu Repositório”):

```
$ svn import /tmp/meuprojeto file:///caminho/do/repositorio/meuprojeto -m "importação i
Adding         /tmp/meuprojeto/branches
Adding         /tmp/meuprojeto/tags
Adding         /tmp/meuprojeto/trunk
Adding         /tmp/meuprojeto/trunk/foo.c
Adding         /tmp/meuprojeto/trunk/bar.c
Adding         /tmp/meuprojeto/trunk/Makefile
…
Committed revision 1.
$
```

Agora o repositório contém esta árvore de dados. Como mencionado anteriormente, você não verá seus dados ao listar diretamente o conteúdo do repositório; os dados estão todos armazenados dentro de uma base de dados. Mas o sistema de arquivos imaginário do repositório agora contém um diretório de alto nível chamado `meuprojeto`, que por sua vez contém seus dados.

Veja que o diretório `/tmp/meuprojeto` original não é alterado; o Subversion sequer tem conhecimento dele. (De fato, você pode até excluir esse diretório se quiser.) Para começar a manipular os dados do repositório, você precisa criar uma nova “cópia de trabalho” dos dados, que são uma espécie de espaço de trabalho particular. Solicite que o Subversion lhe entregue (“*check out*”) uma cópia de trabalho do diretório `meuprojeto/trunk` no repositório:

```
$ svn checkout file:///caminho/do/repositorio/meuprojeto/trunk meuprojeto
A  meuprojeto/foo.c
A  meuprojeto/bar.c
A  meuprojeto/Makefile
…
Checked out revision 1.
```

Agora você tem uma cópia pessoal de uma parte do repositório em um novo diretório chamado `meuprojeto`. Você pode alterar os arquivos em sua cópia de trabalho e então submeter essas alterações de volta para o repositório.

- Entre em sua cópia de trabalho e modifique o conteúdo de algum arquivo.
- Execute **svn diff** para ver uma saída unificada de suas alterações.
- Execute **svn commit** para submeter a nova versão de seu arquivo ao repositório.
- Execute **svn update** para deixar sua cópia de trabalho “atualizada” com o repositório.

Para conhecer todas as coisas que você pode fazer com sua cópia de trabalho, leia Capítulo 2, *Uso Básico*.

Neste ponto, você tem a opção de tornar seu repositório disponível a outras pessoas através de uma rede. Consulte Capítulo 6, *Configuração do Servidor* para aprender sobre os diferentes tipos de processos servidores e sobre como configurá-los.

Apêndice B. Subversion para Usuários de CVS

Este apêndice é um guia para usuários de CVS novos no Subversion. É essencialmente uma lista das diferenças entre os dois sistemas como são “vistos a 10.000 pés de altura”. Em cada seção, nós fornecemos referências a capítulos relevantes, quando possível.

Embora o objetivo do Subversion seja assumir a atual e futura base de usuários do CVS, algumas novas características e mudanças de projeto foram necessárias para corrigir certos comportamentos “quebrados” que o CVS apresentava. Isto significa que, como um usuário de CVS, você pode precisar mudar hábitos—a começar pelos que você esqueceu que eram estranhos.

Os Números de Revisão Agora São Diferentes

No CVS, os números de revisão são por arquivo. Isso porque o CVS armazena seus dados em arquivos RCS; cada arquivo tem um arquivo RCS correspondente no repositório, e o repositório é organizado aproximadamente de acordo com a estrutura da árvore do seu projeto.

No Subversion, o repositório parece um sistema de arquivos único. Cada submissão resulta em uma árvore de sistema de arquivos inteiramente nova; em essência, o repositório é um conjunto ordenado de árvores. Cada uma dessas árvores é rotulada com um número de revisão único. Quando alguém fala sobre a “revisão 54”, está falando sobre uma árvore em particular (e, indiretamente, sobre a forma que o sistema de arquivos apresentava após a 54ª submissão).

Tecnicamente, não é válido falar sobre a “revisão 5 de `foo.c`”. Em vez disso, diria-se “`foo.c` como aparece na revisão 5”. Também seja cuidadoso ao fazer suposições sobre a evolução de um arquivo. No CVS, as revisões 5 e 6 de `foo.c` são sempre diferentes. No Subversion, é mais provável que `foo.c` *não* tenha mudado entre as revisões 5 e 6.

Similarmente, no CVS um rótulo ou ramo é uma anotação no arquivo, ou na informação de versão para aquele arquivo individual, enquanto no Subversion um rótulo ou ramo é uma cópia de uma árvore inteira (por convenção, nos diretórios `/branches` ou `/tags` que aparecem no nível superior do repositório, ao lado de `/trunk`). No repositório como um todo, muitas versões de cada arquivo podem estar visíveis: a última versão em cada ramo, cada versão rotulada, e, claro, a última versão no próprio tronco. Assim, para refinar ainda mais os termos, poderia-se freqüentemente dizer “`foo.c` como aparece em `/branches/REL1` na revisão 5.”

Para mais detalhes sobre este tópico, veja “Revisões”.

Versões de Diretório

O Subversion rastreia estruturas de árvores, e não apenas o conteúdo dos arquivos. Esta é uma das maiores razões pelas quais o Subversion foi escrito para substituir o CVS.

Aqui está o que isto significa para você, como antigo usuário de CVS:

- Os comandos **svn add** e **svn delete** agora funcionam em diretórios, da mesma forma como funcionam em arquivos. O mesmo vale para **svn copy** e **svn move**. Entretanto, estes comandos *não* causam nenhum tipo de mudança imediata no repositório. Em vez disso, os itens de trabalho são simplesmente “marcados” para adição ou exclusão. Nenhuma mudança no repositório acontece até que você execute **svn commit**.

- Os diretórios não são mais simples contêineres; eles têm números de revisão como os arquivos. (Ou, mais propriamente, é correto dizer “diretório `foo/` na revisão 5”.)

Vamos falar mais sobre esse último ponto. O versionamento de diretórios é um problema difícil; como nós queremos permitir cópias de trabalho de revisões mistas, há algumas limitações no quanto podemos abusar deste modelo.

De um ponto de vista teórico, nós definimos que a “revisão 5 do diretório `foo`” significa uma coleção específica de entradas de diretório e propriedades. Agora suponha que começamos a adicionar e remover arquivos de `foo`, e então submetemos. Seria mentira dizer que nós ainda temos a revisão 5 de `foo`. Entretanto, se nós mudássemos o número de revisão de `foo` depois da submissão, isso também seria falso; pode haver outras mudanças em `foo` que nós ainda não recebemos, porque ainda não atualizamos.

O Subversion lida com este problema rastreando secretamente na área `.svn` as adições e exclusões submetidas. Quando você eventualmente executa **svn update**, todas as contas são acertadas com o repositório, e o novo número de revisão do diretório é determinado corretamente. *Portanto, apenas depois de uma atualização é realmente seguro dizer que você tem uma “perfeita” revisão de um diretório.* Na maior parte do tempo, sua cópia de trabalho conterá revisões de diretório “imperfeitas”.

Similarmente, surge um problema se você tenta submeter mudanças de propriedades em um diretório. Normalmente, a submissão mudaria o número de revisão local do diretório de trabalho. Mas, novamente, isso seria falso, porque pode haver adições ou exclusões que o diretório ainda não tem, porque nenhuma atualização aconteceu. *Portanto, não é permitido que você submeta mudanças de propriedade em um diretório, a não ser que ele esteja atualizado.*

Para mais discussão sobre as limitações do versionamento de diretórios, veja “Revisões Locais Mistas”.

Mais Operações Desconectadas

Nos últimos anos, espaço em disco tornou-se ultrajantemente barato e abundante, mas largura de banda não. Portanto, a cópia de trabalho do Subversion foi otimizada em função do recurso mais escasso.

O diretório administrativo `.svn` serve ao mesmo propósito que o diretório `CVS`, exceto porque também armazena cópias somente-leitura e “intocadas” dos seus arquivos. Isto permite que você faça muitas coisas desconectado:

svn status
: Mostra quaisquer mudanças locais que você fez (veja “Obtendo uma visão geral de suas alterações”)

svn diff
: Mostra os detalhes das suas mudanças (veja “Examinando os detalhes de suas alterações locais”)

svn revert
: Remove suas mudanças locais (veja “Desfazendo Modificações de Trabalho”)

Os arquivos originais guardados também permitem que o cliente Subversion envie diferenças ao submeter, o que o CVS não é capaz de fazer.

O último subcomando da lista é novo; ele não apenas removerá mudanças locais, mas irá desmarcar operações agendadas, como adições e exclusões. É a maneira preferida de reverter um arquivo; executar **rm file; svn update** também irá funcionar, mas isso mancha o propósito da atualização. E, por falar nisso…

Distinção Entre Status e Update

No Subversion, nós tentamos dirimir boa parte da confusão entre os comandos **cvs status** e **cvs update**.

O comando **cvs status** tem dois propósitos: primeiro, mostrar ao usuário qualquer modificação local na cópia de trabalho, e, segundo, mostrar ao usuário quais arquivos estão desatualizados. Infelizmente,

devido à dificuldade para ler a saída produzida pelo status no CVS, muitos usuários de CVS não tiram nenhuma vantagem deste comando. Em vez disso, eles desenvolveram um hábito de executar **cvs update** ou **cvs -n update** para ver rapidamente suas mudanças. Se os usuários se esquecem de usar a opção `-n`, isto tem o efeito colateral de fundir alterações no repositório com as quais eles podem não estar preparados para lidar.

Com o Subversion, nós tentamos eliminar esta confusão tornando a saída do **svn status** fácil de ler tanto para humanos quanto para programas analisadores. Além disso, **svn update** só imprime informação sobre arquivos que estão atualizados, e *não* modificações locais.

Status

svn status imprime todos os arquivos que têm modificações locais. Por padrão, o repositório não é contatado. Embora este subcomando aceite um bom número de opções, as seguintes são as mais comumente usadas:

`-u`
: Contatar o repositório para determinar, e então mostrar, informações sobre desatualização.

`-v`
: Mostrar *todas* as entradas sob controle de versão.

`-N`
: Executar não-recursivamente (não descer para os subdiretórios).

O comando **status** tem dois formatos de saída. No formato “curto” padrão, modificações locais parecem com isto:

```
$ svn status
M      foo.c
M      bar/baz.c
```

Se você especificar a opção `--show-updates` (`-u`), um formato mais longo de saída é usado:

```
$ svn status -u
M            1047   foo.c
       *     1045   faces.html
       *            bloo.png
M            1050   bar/baz.c
Status against revision:   1066
```

Neste caso, duas novas colunas aparecem. A segunda coluna contém um asterisco se o arquivo ou diretório está desatualizado. A terceira coluna mostra o número de revisão da cópia de trabalho do item. No exemplo acima, o asterisco indica que `faces.html` seria alterado se nós atualizássemos, e que `bloo.png` é um arquivo recém-adicionado ao repositório. (A falta de qualquer número de revisão próximo a `bloo.png` significa que ele ainda não existe na cópia de trabalho.)

A esta altura, você deve dar uma rápida olhada na lista de todos os códigos de status possíveis em svn status. Aqui estão alguns dos códigos de status mais comuns que você verá:

```
A    O recurso está programado para Adição
D    O recurso está programado para exclusão
M    O recurso tem Modificações locais
C    O recurso tem Conflitos (as mudanças não foram completamente fundidas
       entre o repositório e a versão da cópia de trabalho)
X    O recurso é eXterno a esta cópia de trabalho (pode vir de outro
       repositório).  Veja “Definições Externas”
?    O recurso não está sob controle de versão
```

`!`   `O recurso está faltando ou incompleto (removido por outra ferramenta que não Subversion)`

Para uma discussão mais detalhada de **svn status**, veja “Obtendo uma visão geral de suas alterações”.

Update

svn update atualiza sua cópia de trabalho, e só imprime informação sobre arquivos que ele atualiza.

O Subversion combinou os códigos `P` e `U` do CVS em apenas `U`. Quando ocorre uma fusão ou conflito, o Subversion simplesmente imprime `G` ou `C`, em vez de uma sentença inteira.

Para uma discussão mais detalhada de **svn update**, veja “Atualizando Sua Cópia de Trabalho”.

Ramos e Rótulos

O Subversion não distingue entre espaço do sistema de arquivos e espaço do “ramo”; ramos e rótulos são diretórios normais dentro do sistema de arquivos. Esta é provavelmente o único maior obstáculo mental que um usuário de CVS precisará escalar. Leia tudo sobre isso em Capítulo 4, *Fundir e Ramificar*.

Visto que o Subversion trata ramos e rótulos como diretórios normais, sempre se lembre de efetuar checkout do `tronco` (`http://svn.example.com/repos/calc/trunk/`) do seu projeto, e não do projeto em si (`http://svn.example.com/repos/calc/`). Se você cometer o erro de efetuar checkout do projeto em si, vai terminar com uma cópia de trabalho que contém uma cópia do seu projeto para cada ramo e rótulo que você tem.[1]

Propriedades de Metadados

Uma nova característica do Subversion é que você pode atribuir um metadado arbitrário (ou “propriedades”) as arquivos e diretórios. Propriedades são pares nome/valor arbitrários associados com arquivos e diretórios na sua cópia de trabalho.

Para atribuir ou obter o nome de uma propriedade, use os subcomandos **svn propset** e **svn propget** . Para listar todas as propriedades de um objeto, use **svn proplist**.

Para mais informações, veja “Propriedades”.

Resolução de Conflitos

O CVS marca conflitos com “marcadores de conflito” em linha, e imprime um `C` durante uma atualização. Historicamente, isso tem causado problemas, porque o CVS não está fazendo o suficiente. Muitos usuários esquecem (ou não vêem) o `C` depois que ele passa correndo pelo terminal. Eles freqüentemente esquecem até mesmo que os marcadores de conflitos estão presentes, e então acidentalmente submetem arquivos contendo marcadores de conflitos.

O Subversion resolve este problema tornando os conflitos mais tangíveis. Ele se lembra de que um arquivo encontra-se em um estado de conflito, e não permitirá que você submeta suas mudanças até que execute **svn resolved**. Veja “Resolvendo Conflitos (Combinando Alterações de Outros)” para mais detalhes.

Arquivos Binários e Tradução

No sentido mais geral, o Subversion lida com arquivos binários de forma mais elegante que o CVS. Por usar RCS, o CVS só pode armazenar sucessivas cópias inteiras de um arquivo binário que está sendo alterado. O Subversion, entretanto, expressa as diferenças entre arquivos usando um algoritmo de diferenciação binária, não importando se eles contêm dados textuais ou binários. Isso significa que todos os arquivos são armazenados diferencialmente (comprimidos) no repositório.

Os usuários de CVS têm que marcar arquivos binários com flags `-kb`, para prevenir que os dados sejam corrompidos (devido a expansão de palavras-chave e tradução de quebras de linha). Eles algumas vezes se esquecem de fazer isso.

O Subversion segue a rota mais paranóica—primeiro, nunca realiza nenhum tipo de tradução de palavra-chave ou de quebra de linha, a menos que você explicitamente o instrua a fazê-lo (veja "Substituição de Palavra-Chave" e "Seqüência de Caracteres de Fim-de-Linha" para mais detalhes). Por padrão, o Subversion trata todos os dados do arquivo como cadeias de bytes literais, e os arquivos sempre são armazenados no repositório em estado não-traduzido.

Segundo, o Subversion mantém uma noção interna de se um arquivo contém dados "de texto" ou "binários", mas esta noção existe *apenas* na cópia de trabalho. Durante um **svn update**, o Subversion realizará fusões contextuais em arquivos de texto modificados localmente, mas não tentará fazer o mesmo com arquivos binários.

Para determinar se uma fusão contextual é possível, o Subversion examina a propriedade `svn:mime-type`. Se o arquivo não tem a propriedade `svn:mime-type`, ou tem um mime-type que é textual (por exemplo, `text/*`), o Subversion supõe que ele é texto. Caso contrário, o Subversion supõe que o arquivo é binário. O Subversion também ajuda os usuários executando um algoritmo para detectar arquivos binários nos comandos **svn import** e **svn add**. Estes comandos farão uma boa suposição e então (possivelmente) colocarão uma propriedade `svn:mime-type` binária no arquivo que está sendo adicionado. (Se o Subversion fizer uma suposição errada, o usuário sempre pode remover ou editar manualmente a propriedade.)

Módulos sob Controle de Versão

Diferentemente do que ocorre no CVS, uma cópia de trabalho do Subversion sabe que efetuou checkout de um módulo. Isso significa que se alguém muda a definição de um módulo (por exemplo, adiciona ou remove componentes), então uma chamada a **svn update** irá atualizar a cópia de trabalho apropriadamente, adicionando e removendo componentes.

O Subversion define módulos como uma lista de diretórios dentro de uma propriedade de diretório: veja "Definições Externas".

Autenticação

Com o pserver do CVS, exige-se que você "inicie sessão" no servidor antes de qualquer operação de leitura ou escrita—às vezes você tem de iniciar uma sessão até para operações anônimas. Com um repositório Subversion usando Apache **httpd** ou **svnserve** como servidor, você não fornece quaisquer credenciais de autenticação a princípio —se uma operação que você realiza requer autenticação, o servidor irá pedir suas credenciais (sejam essas credenciais nome de usuário e senha, um certificado de cliente, ou mesmo ambos). Assim, se o seu repositório pode ser lido por todo o mundo, não será exigido que você se autentique para operações de leitura.

Assim como o CVS, o Subversion ainda guarda suas credenciais em disco (em seu diretório `~/.subversion/auth/`), a menos que você o instrua a não fazê-lo, usando a opção `--no-auth-cache`.

A exceção a este comportamento, entretanto, é no caso de se acessar um servidor **svnserve** através de um túnel SSH, usando o esquema de URL `svn+ssh://`. Nesse caso, o programa **ssh** incondicionalmente requer autenticação para iniciar o túnel.

Convertendo um Repositório de CVS para Subversion

Talvez a forma mais importante de familiarizar usuários de CVS com o Subversion é deixá-los continuar trabalhando em seus projetos usando o novo sistema. E mesmo que isso possa de certa forma ser

conseguido usando uma importação simples em um repositório Subversion de um repositório CVS exportado, a solução mais completa envolve transferir não apenas o estado mais recente dos seus dados, mas toda a história atrás dele também, de um sistema para o outro. Isto é um problema extremamente difícil de resolver, que envolve deduzir conjuntos de mudanças na falta de atomicidade, e traduzir entre as políticas de ramificação completamente ortogonais dos dois sistemas, entre outras complicações. Todavia, há um punhado de ferramentas prometendo suportar ao menos parcialmente a habilidade de converter repositórios CVS em repositórios Subversion.

A mais popular (e provavelmente a mais madura) ferramenta de conversão é cvs2svn (http://cvs2svn.tigris.org/), um script Python originalmente criado por membros da própria comunidade de desenvolvimento do Subversion. Esta ferramenta é destinada a ser executada exatamente uma vez: ela examina seu repositório CVS diversas vezes e tenta deduzir submissões, ramos e rótulos da melhor forma que consegue. Quando termina, o resultado é ou um repositório Subversion ou um arquivo de despejo portável representando a história do seu código. Veja o website para instruções detalhadas e advertências.

Apêndice C. WebDAV e Autoversionamento

WebDAV é uma extensão do HTTP, e está se tornando cada vez mais popular como um padrão para compartilhamento de arquivos. Hoje em dia, os sistemas operacionais estão se tornado extremamente relacionados à Web, e muitos agora têm suporte para montar “compartilhamentos” exportados por servidores WebDAV.

Se você usa o Apache como seu servidor de rede para o Subversion, então para algumas extensões você também deve estar executando um servidor WebDAV. Este apêndice oferece algum suporte sobre a natureza deste protocolo, como o Subversion o utiliza, e o quão boa é a interoperabilidade do Subversion com outros softwares relacionados com o WebDAV.

O que é WebDAV?

DAV é uma sigla em inglês para “Versionamento e Autoração Distribuída”. A RFC 2518 define um conjunto de conceitos e métodos de extensão relacionados a HTTP 1.1 que transformam a web em uma mídia mais universal de leitura e escrita. A idéia básica é que um servidor web que siga o padrão WebDAV pode agir como um servidor de arquivos genérico; os clientes podem “montar” pastas compartilhadas sobre HTTP que se comportem como qualquer outro sistema de arquivos de rede (como NFS ou SMB.)

O problema, porém, é que apesar do acrônimo, a especificação RFC atualmente não descreve qualquer forma de controle de versão. Clientes básicos de WebDAV e servidores assumem apenas uma versão para cada arquivo ou diretório existente, e podem ser repetidamente sobrescritos.

Devido a RFC 2518 não contemplar conceitos de versionamento, outro comitê tomou para si a responsabilidade de escrever a RFC 3253 alguns anos depois. A nova RFC adiciona conceitos de versionamento ao WebDAV, fazendo valer o significado do “V” em “DAV”—daí o termo “DeltaV”. Clientes e servidores WebDAV/DeltaV quase sempre são apenas chamados de programas “DeltaV”, uma vez que DeltaV implica na existência de WebDAV básico.

O padrão WebDAV original tem sido largamente usado com sucesso. Cada sistema operacional de computadores modernos tem uma implementação interna de um cliente WebDAV (mais detalhes a seguir), e diversas aplicações independentes populares também são capazes de se comunicar em WebDAV—como o Microsoft Office, Dreamweaver, e Photoshop para relacionar apenas algumas. No lado do servidor, o servidor web Apache tem capacidade de oferecer serviços WebDAV desde 1998 e é considerado o padrão open-source “de facto”. Há outros servidores WebDAV disponíveis, incluindo o próprio IIS da Microsoft.

Já o DeltaV, infelizmente, não tem sido tão bem sucedido. É muito difícil de encontrar clientes ou servidores DeltaV. Os poucos que existem são produtos comerciais relativamente desconhecidos, e assim é muito difícil testar interoperabilidade. Não é totalmente claro o porquê de que o DeltaV permanece estagnado. Alguns argumentam apenas que a especificação é muito complexa, já outros dizem que apesar de os recursos do WebDAV terem apelo maciço (ao menos para usuários técnicos que apreciem compartilhamento de arquivos em rede), recursos de controle de versão não são interessantes ou mesmo necessários para a maioria dos usuários. Finalmente, alguns ainda consideram que o DeltaV permanece impopular porque ainda não há um servidor como produto open-source que o implemente tão bem.

Ainda que o Subversion ainda esteja em fase de desenvolvimento, parece uma grande idéia usar o Apache como servidor de rede. Ele já possui um módulo para prover serviços WebDAV. DeltaV era uma especificação relativamente nova. A esperança era que o módulo servidor do Subversion (o **mod_dav_svn**) pudesse eventualmente evoluir para uma implementação open-source de referência DeltaV. Infelizmente, o DeltaV tem um modelo de versionamento muito específico que não é exatamente adequado ao modelo do Subversion. Alguns conceitos foram mapeados, mas outros não.

Mas então, o que isto significa?

Primeiro, o cliente Subversion não é uma implementação completa de um cliente DeltaV. Ele precisa de algumas certas coisas do servidor que o DeltaV em si não pode prover, e assim ele é altamente dependente de diversas requisições HTTP `REPORT` específicas para o Subversion que apenas o **mod_dav_svn** pode entender.

Segundo, o **mod_dav_svn** também não é um servidor DeltaV completamente implementado. Muitas partes da especificação DeltaV eram irrelevantes para o Subversion, e assim simplesmente não foram implementadas.

Ainda há um certo debate na comunidade de desenvolvedores se ainda é ou não adequado se preocupar em tentar remediar estas situações. É quase impensável alterar-se o projeto do Subversion para corresponder à especificação DeltaV, então provavelmente não há nada que que um cliente não pode aprender para ter tudo o que precisa de um servidor DeltaV. Por outro lado, o **mod_dav_svn** *poderia* ser desenvolvido para implementar tudo do DeltaV, mas é difícil encontrar motivação para fazê-lo—não há muitos clientes DeltaV com os quais se possa comunicar.

Autoversionamento

Ainda que o cliente Subversion não seja um cliente DeltaV completo, e que nem mesmo o servidor Subversion serja um servidor DeltaV completo, ainda há um lampejo de interoperabilidade com o WebDAV com o qual se contentar: é o chamado autoversionamento.

Autoversionamento é um recurso opcional definido no padrão DeltaV. Um servidor DeltaV típico irá rejeitar um cliente WebDAV ignorante que tente fazer um `PUT` em um arquivo que esteja sob controle de versão. Para alterar um arquivo versionado, o servidor espera uma série de requisições próprias relacionadas a versionamento: algo como `MKACTIVITY`, `CHECKOUT`, `PUT`, `CHECKIN`. Mas se o servidor DeltaV suportar autoversionamento, então as requisições de escrita a partir de clientes WebDAV básicos são aceitas. O servidor se comporta como se o cliente *tivesse* feito o conjunto de requisições próprias relacionadas ao versionamento adequadas, executando uma submissão (*commit*) por debaixo dos panos. Em outras palavras, ele permite a um servidor DeltaV interoperar com clientes WebDAV ordinários que não entendam versionamento.

Pelo fato de que muitos sistemas operacionais já têm clientes WebDAV integrados, o caso de uso para este recurso pode ser incrivelmente interessante a administradores que estejam trabalhando com usuários não-técnicos: imagine um escritório de usuários comuns executando Microsoft Windows ou Mac OS. Cada usuário “monta” o repositório Subversion, o qual aparece como uma pasta de rede normal. Eles usam a pasta compartilhada como estão acostumados: abrem arquivos, fazem modificações, salvam. Enquanto isso, o servidor está automaticamente versionando tudo. Qualquer administrador (ou usuário com conhecimento adequado) ainda pode usar um cliente Subversion para pesquisar num histórico e obter versões mais antigas dos dados.

Este cenário não é fictício: é real e funciona, sendo o Subversion versão 1.2 ou posterior. Para ativar o autoversionamento no **mod_dav_svn**, use a diretiva `SVNAutoversioning` dentro do bloco Location no `httpd.conf`, mais ou menos assim:

```
<Location /repos>
  DAV svn
  SVNPath /path/to/repository
  SVNAutoversioning on
</Location>
```

Quando o SVNAutoversioning está ativa, requisições de escrita a partir de clientes WebDAV resultam em submissões automáticas. Uma mensagem de log genérica é gerada automaticamente e anexada a cada revisão.

Antes de ativar este recurso, no entanto, entenda no que você está se metendo. Clientes WebDAV tendem a fazer *muitas* requisições de escrita, resultando em um número enorme de revisões

submetidas automaticamente. Por exemplo, ao salvar dados, muitos clientes farão um PUT de um arquivo de zero bytes (como uma forma de reservar um nome) seguida de outro PUT com os dados do arquivo real. Um único salvamento de arquivo resulta em duas submissões em separado. Também considere que muitas aplicações realizam salvamento automático em intervalos definidos, resultando em ainda mais submissões de rede.

Se você tiver um script de hook post-commit que envie um e-mail, você pode querer desabilitar a geração de e-mail como um todo, ou em certas seções do repositório; depende de como você acha que o fluxo de e-mails resultante ainda sejam notificações importantes ou não. Ainda, um script de hook post-commit esperto pode diferenciar entre uma transação criada a partir de autoversionamento e uma criada a partir de um **svn commit** normal. O truque é olhar para uma propriedade da revisão chamada `svn:autoversioned`. Se existir, o commit foi feito por um cliente WebDAV genérico.

Outro recurso que pode ser um útil complemento para o `SVNAutoversioning` vem do módulo `mod_mime` do Apache. Se um cliente WebDAV adicionar um novo arquivo ao repositório, não haverá uma oportunidade para que o usuário defina a propriedade `svn:mime-type`. Isto pode fazer com que o arquivo apareça como um ícone genérico quando visto em uma pasta WebDAV compartilhada, não associado a nenhuma aplicação. Uma solução é ter um administrador de sistema (ou outra pessoa com conhecimento do Subversion) que obtenha uma cópia de trabalho e defina manualmente a propriedade `svn:mime-type` nos arquivos necessários. Mas potencialmente essas tarefas de limpeza potencialmente não têm fim. Ao invés disso, você pode usar a diretiva `ModMimeUsePathInfo` no bloco `<Location>` de seu Subversion:

```
<Location /repos>
  DAV svn
  SVNPath /path/to/repository
  SVNAutoversioning on

  ModMimeUsePathInfo on

</Location>
```

Esta diretiva permite que o `mod_mime` para tentar deduzir automaticamente o tipo mime em novos arquivos adicionados ao repositório pelo autoversionamento. O módulo verifica a extensão do nome do arquivo e possivelmente seu conteúdo também; se o arquivo corresponder a alguns padrões comuns, então a propriedade `svn:mime-type` será definida automaticamente.

Interoperabilidade com Softwares Clientes

Todos os clientes WebDAV entram em uma de três categorias—aplicações independentes, extensões para gerenciadores de arquivos, ou implementações de sistemas de arquivos. Estas categorias definem amplamente os tipos de funcionalidades WebDAV disponíveis aos usuários. Tabela C.1, “Clientes WebDAV Comuns” mostra nossa categorização e uma breve descrição de algumas das partes mais comuns de softwares disponíveis para WebDAV. Mais detalhes sobre o que esses softwares oferecem, bem como sua categorização geral, podem ser encontrados nas seções seguintes.

Tabela C.1. Clientes WebDAV Comuns

Software	Tipo	Windows	Mac	Linux	Descrição
Adobe Photoshop	Aplicação WebDAV independente	X			Software de edição de imagens, que permite abrir, e escrever diretamente em URLs WebDAV

Software	Tipo	Windows	Mac	Linux	Descrição
Cadaver	Aplicação WebDAV independente		X	X	Cliente WebDAV de linha de comando que suporta transferência de arquivos, árvores, e operações de travamento
DAV Explorer	Aplicação WebDAV independente	X	X	X	Ferramenta GUI Java para navegação em compartilhamentos WebDAV
Macromedia Dreamweaver	Aplicação WebDAV independente	X			Software de produção web capaz de ler de e escrever diretamente em URLs WebDAV
Microsoft Office	Aplicação WebDAV independente	X			Suíte de produtividade de escritório com diversos componentes capazes de ler de e escrever diretamente em URLs WebDAV
Microsoft Web Folders	Extensão WebDAV para gerenciador de arquivos	X			Programa GUI para gerenciador de arquivos que permite operações de árvore em compartilhamentos WebDAV
GNOME Nautilus	Extensão WebDAV para gerenciador de arquivos			X	Gerenciador de arquivos visual capaz de executar operações de árvore em compartilhamentos WebDAV
KDE Konqueror	Extensão WebDAV para gerenciador de arquivos			X	Gerenciador de arquivos visual capaz de executar operações de árvore em

Software	Tipo	Windows	Mac	Linux	Descrição
					compartilhamentos WebDAV
Mac OS X	Implementação de sistema de arquivos WebDAV		X		Sistema operacional com suporte nativo para montagem de compartilhamentos WebDAV
Novell NetDrive	Implementação de sistema de arquivos WebDAV	X			Programa de mapeamento de rede para associar letras de um drive Windows a um compartilhamento WebDAV remoto
SRT WebDrive	Implementação de sistema de arquivos WebDAV	X			Software de transferência de arquivos que, além de outras coisas, permite associação de letras de drives Windows a compartilhamentos WebDAV remotos
davfs2	Implementação de sistema de arquivos WebDAV			X	Driver para sistema de arquivos do Linux que lhe permite montar compartilhamentos WebDAV

Aplicações WebDAV Independentes

Uma aplicação WebDAV é um programa capaz de conversar via protocolos WebDAV com um servidor WebDAV. Vamos abordar alguns dos programas mais populares com esse tipo de suporte a WebDAV.

Microsoft Office, Dreamweaver, Photoshop

Para Windows, há diversas aplicações bem conhecidas que já possuem funcionalidade de cliente WebDAV, tais como os programas Microsoft Office, [1] Adobe Photoshop, e Macromedia Dreamweaver. Eles são capazes de abrir e salvar URLs diretamente, e tendem a fazer uso massivo de travas WebDAV ao editar arquivos.

Veja que ainda que muitos desses programas também existam para o Mac OS X, eles não parecem suportar o WebDAV nativamente nessa plataforma. De fato, no Mac OS X, a caixa de diálogo File-

[1] O suporte a WebDAV foi removido do Microsoft Access por algum motivo, mas existe nos demais programas da suíte de escritório.

>Open não permite de forma alguma que você digite um caminho ou uma URL. É como se os recursos para WebDAV tivessem sido deliberadamente removidos das versões Macintosh desses programas, uma vez que o sistema de arquivos do OS X já provê um excelente suporte de baixo nível para WebDAV.

Cadaver, DAV Explorer

Cadaver é um programa de linha de comando muito enxuto do Unix para navegação e realização de modificações em compartilhamentos WebDAV. Como o cliente do Subversion, ele usa a biblioteca HTTP neon—sem nenhuma surpresa, uma vez que tanto o neon quanto o cadaver são desenvolvidos pelo mesmo autor. O Cadaver é um software livre (licença GPL) e está disponível em http://www.webdav.org/cadaver/.

Usar o cadaver se parece com usar um programa FTP de linha de comando, sendo assim extremamente útil para depuração básica de WebDAV. O programa pode ser usado para se fazer upload ou download de arquivos de uma só vez, e também examinar propriedades, além de copiar, mover, travar ou destravar arquivos:

```
$ cadaver http://host/repos
dav:/repos/> ls
Listing collection `/repos/': succeeded.
Coll: > foobar                                 0  May 10 16:19
      > playwright.el                       2864  May  4 16:18
      > proofbypoem.txt                     1461  May  5 15:09
      > westcoast.jpg                      66737  May  5 15:09

dav:/repos/> put README
Uploading README to `/repos/README':
Progress: [=============================>] 100.0% of 357 bytes succeeded.

dav:/repos/> get proofbypoem.txt
Downloading `/repos/proofbypoem.txt' to proofbypoem.txt:
Progress: [=============================>] 100.0% of 1461 bytes succeeded.
```

DAV Explorer é outro cliente WebDAV independente, escrito em Java. É distribuído sob uma licença livre ao estilo da do Apache e está disponível em http://www.ics.uci.edu/~webdav/. O DAV Explorer faz tudo o que o cadaver faz, mas tem as vantagens de ser uma aplicação gráfica mais portável e mais amigável. Também foi um dos primeiros clientes a dar suporte ao novo protocolo de acesso WebDAV (RFC 3744).

Obviamente, o suporte a ACL do DAV Explorer é inútil neste caso, já que o **mod_dav_svn** não dá suporte a ele. O fato de tanto o Cadaver quanto o DAV Explorer suportarem alguns comandos DeltaV, particularmente, também é igualmente inútil, uma vez que eles não permitem requisições `MKACTIVITY`. Mas de qualquer forma isso também não é relevante; estamos assumindo que todos esses clientes trabalham normalmente com um repositório com autoversionamento.

Extensões WebDAV para gerenciadores de arquivos

Alguns programas gráficos gerenciadores de arquivos dão suporte a extensões WebDAV que permitem ao usuário navegar em compartilhamentos DAV como se fossem apenas mais um diretório no computador local, além de realizar operações básicas de edição de árvore nos itens naquele compartilhamento. Por exemplo, o Windows Explorer é capaz de navegar em um servidor WebDAV como um “local de rede”. Os usuários podem arrastar arquivos de e para sua área de trabalho, ou podem renomear, copiar, ou excluir arquivos normalmente. Mas pelo fato de ser apenas um recurso do gerenciador de arquivos, o compartilhamento DAV não é visível às aplicações ordinárias. Todas as interações DAV devem acontecer dentro da interface do gerenciador de arquivos.

Microsoft Web Folders

A Microsoft foi uma das incentivadoras originais da especificação WebDAV, e começou desenvolvendo um cliente no Windows 98, conhecido como “Web Folders”. Este cliente também estava presente no Windows NT4 e no 2000.

O cliente Web Folders original era uma extensão para o Explorer, o principal programa gráfico usado para navegar em sistemas de arquivos. Ele funciona suficientemente bem. No Windows 98, o recurso precisa ser explicitamente instalado se Web Folders já não estiverem visíveis dentro do “Meu Computador”. No Windows 2000, simplesmente adicione um novo “local de rede”, informe a URL, e o compartilhamento surgirá para navegação.

Com o lançamento do Windows XP, a Microsoft começou a distribuir uma nova implementação do Web Folders, conhecida como “WebDAV mini-redirector”. A nova implementação é um cliente em nível de sistema de arquivos, que permite que compartilhamentos WebDAV sejam montados como letras de drives. Infelizmente, esta implementação é incrivelmente problemática. O cliente frequentemente tenta converter URLs http (`http://host/repos`) para a notação UNC de compartilhamentos (`\\host\repos`); ele também tenta usar a autenticação de domínio do Windows para responder aos desafios de autenticação basic do HTTP, enviando nomes de usuários como `HOST\username`. Estes problemas de interoperabilidade são críticos e estão documentados em diversos locais pela web, para frustração de muitos usuários. Mesmo Greg Stein, o autor original do módulo WebDAV do Apache, desaconselha o uso do Web Folders do XP para comunicação com um servidor Apache.

Vê-se, porém, que a implementação original do Web Folders “somente para Windows Explorer” não morreu no XP, mas só está enterrada. Ainda é possível encontrá-la usando esta técnica:

1. Vá em 'Locais de Rede'.
2. Adicione um novo local de rede.
3. Quando solicitado, informe a URL do repositório, mas *inclua um número de porta* na URL. Por exemplo, `http://host/repos` deveria ser informado como `http://host:80/repos`.
4. Responda a quaisquer solicitações de autenticação.

Há alguns outros rumores sobre formas de contornar outros problemas, mas nenhum deles parece funcionar em todas as versões e com os diferentes patches do Windows XP. Em nossos testes, apenas estes passos anteriores parecem funcionar consistentemente em cada sistema. É consenso geral da comunidade WebDAV que você deveria evitar essa nova implementação de Web Folders e usar a antiga no lugar, e que se você precisar de um cliente em nível de sistema de arquivos para o Windows XP, use então um programa de terceiros como o WebDrive ou o NetDrive.

Uma última dica: se você estiver tentando usar o XP Web Folders, certifique-se de ter realmente a sua versão mais recente da Microsoft. Por exemplo, a Microsoft disponibilizou uma versão com *bugs* corrigidos em janeiro de 2005, que encontra-se disponível em http://support.microsoft.com/?kbid=892211. Em particular, esta versão é conhecida por corrigir um problema em que ao se navegar em um compartilhamento DAV resultava em uma inesperada recursão infinita.

Nautilus, Konqueror

O Nautilus é o gerenciador de arquivos/navegador oficial do ambiente GNOME (http://www.gnome.org), já o Konqueror é o gerenciador de arquivos/navegador para o ambiente KDE (http://www.kde.org). Estas duas aplicações já possuem nativamente o suporte a WebDAV em nível de sistema de arquivos, e trabalham muito bem com repositórios com recurso de autoversionamento.

No Nautilus do GNOME, a partir do menu File, escolha Open location e entre com a URL. O repositório deverá ser então exibido como um sistema de arquivos como outro qualquer.

No Konqueror do KDE, você precisa usar o esquema `webdav://` ao informar uma URL na barra de endereços. Se você informar uma URL `http://`, o Konqueror vai se comportar como um

navegador web comum. Você provavelmente verá a listagem genérica do diretório HTML produzida pelo **mod_dav_svn**. Ao informar `webdav://host/repos` ao invés de `http://host/repos`, faz com que o Konqueror atue como cliente WebDAV e exiba o repositório como um sistema de arquivos.

Implementações de sistemas de arquivos WebDAV

Uma implementação de sistemas de arquivos WebDAV é sem dúvida o melhor tipo de cliente WebDAV. É implementada como um módulo de sistema de arquivos em baixo nível, normalmente já embutido no próprio kernel. Isto quer dizer que o compartilhamento DAV é montado como qualquer outro sistema de arquivos de rede, semelhante a montagem de um compartilhamento NFS no Unix, ou o mapeamento de um compartilhamento SMB em uma letra de drive no Windows. Como resultado, este tipo de cliente oferece acesso de leitura/escrita a WebDAV de forma completamente transparente para todos os programas. As aplicações nem sequer se dão conta de que requisições WebDAV estão acontecendo.

WebDrive, NetDrive

Ambos, WebDrive e NetDrive, são excelentes produtos comerciais que permitem que um compartilhamento WebDAV seja mapeado como letras de drives no Windows. Nunca tivemos nada além de êxito com estes produtos. No momento em que este livro estava sendo escrito, o WebDrive podia ser adquirido junto a South River Technologies (http://www.southrivertech.com). Já o NetDrive vem junto com o Netware, sem custos adicionais, e pode ser encontrado na web procurando-se por “netdrive.exe”. Ainda que esteja livremente disponível na internet, os usuários precisam ter uma licença do Netware para usá-lo. (Se você acha isso estranho, você não está sozinho. Veja esta página no site da Novell: http://www.novell.com/coolsolutions/qna/999.html)

Mac OS X

O sistema operacional OS X da Apple tem um cliente WebDAV integrado em nível de sistema de arquivos. A partir do Finder, selecione o item Connect to Server a partir do menu Go. Entre com uma URL WebDAV, e ela aparecerá como um disco em sua área de trabalho, tal como qualquer outro volume montado. Você também pode montar um compartilhamento WebDAV a partir do terminal do Darwin usando o tipo de sistema de arquivos `webdav` com o comando **mount**:

```
$ mount -t webdav http://svn.example.com/repos/project /some/mountpoint
$
```

Note que se seu **mod_dav_svn** for anterior à versão 1.2, o OS X vai se negar a montar o compartilhamento como leitura-escrita; e ele aparecerá como somente-leitura. Isto é porque o OS X insiste em usar travar para o suporte a compartilhamentos leitura-escrita, e o recurso de travas de arquivos só surgiu no Subversion versão 1.2.

Mais uma palavra de alerta: o cliente WebDAV do OS X algumas vezes pode ser excessivamente sensível a redirecionamentos HTTP. Se o OS X não for capaz de montar um repositório como um todo, você pode precisar habilitar a diretiva BrowserMatch no `httpd.conf` de seu servidor Apache:

```
BrowserMatch "^WebDAVFS/1.[012]" redirect-carefully
```

Linux davfs2

Linux davfs2 é um módulo de sistema de arquivos para o kernel do Linux, cujo desenvolvimento está centrado em http://dav.sourceforge.net/. Uma vez instalado este módulo, um compartilhamento pode ser montado normalmente com o comando mount do Linux:

```
$ mount.davfs http://host/repos /mnt/dav
```

Apêndice D. Ferramentas de Terceiros

O projeto modular do Subversion (abordado em “Projeto da Biblioteca em Camadas”) e a disponibilidade de extensões para outras linguagens (como descrito em “Usando Outras Linguagens além de C e C ++”) o tornam um provável candidato a ser usado como uma extensão ou como suporte (*backend*) para outros componentes de software. Para uma listagem de diversas ferramentas de terceiros que estão elas mesmas fazendo uso de funcionalidades do Subversion, confira a página de Links no site do Subversion (http://subversion.tigris.org/project_links.html).

Apêndice E. Copyright

b. to create and reproduce Derivative Works;

c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

e.

For the avoidance of doubt, where the work is a musical composition:

i. Performance Royalties Under Blanket Licenses. Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.

ii. Mechanical Rights and Statutory Royalties. Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

f. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions.The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of

warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any reference to such Licensor or the Original Author, as requested.

b. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this

License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Índice Remissivo